

Many-Core Processor Architectures for Topographic Computations

Beyond Parallelism

Ákos Zarándy

Doctoral Dissertation

for the Doctor of the Hungarian Academy of Sciences degree

Budapest

2009

Table of contents

1 INTRODUCTION	2
2 DIRECT CNN TEMPLATE DESIGN	10
2.1 The Cellular Nonlinear/Neural Network model	11
2.2 Uncoupled CNN templates	14
2.2.1 Binary-input \rightarrow binary-output uncoupled CNN templates	16
2.3 Coupled CNN templates	24
2.3.1 Application range	34
2.4 Conclusions	36
3 VIRTUAL PROCESSOR ARRAYS	38
3.1 The virtual topographic array concept	38
3.1.1 Virtual processor arrays for early vision applications	39
3.1.2 Virtual processor arrays using foveal approach	41
3.2 Mixed-signal virtual processor array architecture for analog video signal processing ...	42
3.2.1 Timing details	44
3.2.2 Processor options	45
3.3 Pipe-line virtual digital physical processor array for high resolution image processing.	45
3.3.1 Principles of operation	46
3.3.2 Architecture description	48
3.3.3 Program flow considerations	57
3.4 Bi-i, a foveal processor architecture based camera system.....	58
3.4.1 Low resolution (128×128) ultra high speed mode of the Bi-i	61
3.4.2 High resolution (megapixel) video speed mode of the Bi-i.....	62
3.4.3 Virtual high resolution (megapixel) high speed mode of the Bi-i.....	62
3.5 VISCUBE, a foveal processor architecture based vision chip.....	63
3.5.1 VISCUBE architecture.....	63
3.5.2 Data communication, conversion, scaling.....	69
3.5.3 Operation, control and synchronization.....	70
3.5.4 Target algorithms: registration	71
3.6 Conclusions.....	71
4 LOW-POWER PROCESSOR ARRAY DESIGN STRATEGY FOR SOLVING COMPUTATIONALLY INTENSIVE 2D TOPOGRAPHIC PROBLEMS.....	74

4.1 Architecture descriptions	75
4.1.1 <i>Classic DSP-memory architecture</i>	75
4.1.2 <i>Pipe-line architectures</i>	77
4.1.3 <i>Coarse-grain cellular parallel architectures</i>	79
4.1.4 <i>Fine-grain fully parallel cellular architectures with discrete time processing</i>	81
4.1.5 <i>Fine-grain fully parallel cellular architecture with continuous time processing</i>	82
4.2 Implementation and efficiency analysis of various operators.....	83
4.2.1 <i>Categorization of 2D operators</i>	83
4.2.2 <i>Processor utilization efficiency of the various operation classes</i>	87
4.2.3 <i>Multi-scale processing</i>	94
4.3 Comparison of the architectures	95
4.4 Optimal architecture selection	100
4.5 Summary	103
4.6 Conclusions.....	104
ACKNOWLEDGEMENT	106
REFERENCES	108
APPENDIX: DESCRIPTION OF THE CITED CNN TEMPLATES.....	II

1 Introduction

The first 8 bit microprocessor, the Intel 8080 applied about 6000 transistors. It ran on 2MHz, and provided 0.5 MOps computational power on 8 bit integers. Following the pace dictated by Moore's law, the latest microprocessors uses three billion transistors on a single chip nowadays. Their clock frequency goes up to 4 GHz, and they provide 50GFlops on 32 bit floating point numbers. Assuming that 1 Flop @ 32 bit is roughly equivalent with 10 Ops @ 8 bit, we get that a high-end processor nowadays with the given parameters is 1,000,000 (one million) times more powerful than the i8080. This is an amazing technical success indeed! However, it is worth to calculate the efficiency of the technological usage.

If we look at the transistor counts, we can see that roughly 500,000 pieces of 8 bit microprocessors can be implemented on a single chip nowadays. Moreover, the technology enables to drive them on at least 2000 times higher clock frequency. This means that the technological development offered 1,000,000,000 (one billion) times speed up, and as we have seen, 1,000,000 (one million) times was utilized out of it "only". The missing 1000 time speedup is very huge, because that is the result of about 15 years of CPU development. How can at least a part of this missing 1000 times performance increase be utilized?

Processor architecture designers are focusing their attention to many core architectures, because neither the further widening of the word length of the super-scalar processors nor the further increase of the clock frequency work due to low efficiency of the formal and very high power consumption penalties of the latter. This initialized a transition in the processor industry towards the multi-core designs. A new Moore's law estimates that the number of the cores in a single chip doubles in each year [56]. Intel and AMD came out with the duals [60][61] and later the quads for desktops, while IBM and SUN built 9 core processors [58][62] for servers. Other designs reached close to 100 processor cores [63][66]. Nvidia introduced the CUDA processor array family, which goes up to 240 cores [65].

Though this is a very impressive roadmap, we have to know, that there are major problems with the many-core architectures, because the gigantic mass of the nowadays used software, what we would like to use in the future also, cannot be efficiently executed on them. Moreover, there is no general solution how to modify an algorithm to make it efficiently executed on a many-core device. On the other hand, the optimal many-core processor architecture selection for a given problem in general is also unknown. And top of all that, none has an idea what we can do with ten thousand or even hundred thousand processor cores on a single die, though the readily available CMOS technology makes the implementation of such a mega-processor array absolutely feasible. The answers to these architectural and algorithmic questions are one of the most intensively researched areas of the field, because they will shape the short and midterm development paces of the computer technology.

In my Dissertation I am addressing one segment of these problems, namely the topographic many core processor architectures, and their application in 2D data array (image) processing. The special feature of these topographic processor arrays is that the processor cores are arranged to the vertexes of a regular grid. This regularity introduces novel phenomena to these processor arrays, namely the appearing physical address of the individual processors and the increasing precedence of the locality (the local interconnectivity). This means that the communication between the neighboring cores becomes much cheaper than the communication among farther cores. This leads to the efficient implementation of the wave-type operators (see later) on these topographic architectures, because the processor boundaries do not raise barriers for the propagating wave-fronts.

One of these architectures is the Cellular Neural/nonlinear Network (CNN). CNN is based on a large number of locally interconnected, identical, programmable, mixed-signal processors cells, arranged to a rectangular grid. The novelty of these devices is, that the cells are dynamical elements, and the aggregate temporal cell dynamics over the array generates spatial-temporal wave phenomena (propagating waves). The array dynamics of these topographic processors introduces operators, which are beyond the Boolean logic [31]. These operators are spatial-temporal dynamic phenomena and their operands are entire images rather than a few scalars. Typical operators from this group are the various feedback convolutions, the diffusion, the global average applied to a whole image, the global OR applied to an entire binary field [42], a single transient centroid, grassfire, or skeleton [48] operation. When the CNN dynamics is embedded in a stored programmable machine (CNN Universal Machine [27]) we can start thinking in spatial-temporal algorithms.

Other special feature of the CNN processor chip is that the data is topographically represented on the processors on a one to one manner. This one-to-one correspondence between pixels and processors makes possible to add sensors to each pixel, which converts the device to a sensor-processor array. The compactness of the mixed-signal processor cores enables to implement relatively large sensor-processor arrays. The two largest are the 128×128 sized ACE16k sensor-processor array [42] containing over 16,000 processor cells (cores), and the 176×144 sized Q-Eye [67].

Though CNN-UM (or “CNN computer”) [27] is a fully programmable Turing machine equivalent device [28], its programming is far from triviality. Programmers have to create templates, which describe the interconnection weights of a dynamically coupled, 2D dynamic processor array in such a way that the trajectories of the resulting coupled differential state equation system lead to the desired output of the particular image processing function. Due to its dynamic coupling, CNN can implement spatial-temporal wave phenomena, including propagating binary waves sweeping through the entire array. In my first thesis, I am introducing a template design method for non-propagating and propagating type binary input-binary output operators.

In the last few years numerous locally interconnected massively parallel low-power topographic sensor-processor arrays have been designed both in the academic [42][44][49] and in the commercial [67][68] arena. The common features of them is that all of them is designed to process medium resolution images on very high speed, but they cannot handle high (VGA or megapixel) resolution image flows on video speed. This is due to the lack of enough on-chip memory required by the original 2D processor architecture, rather than the lack of computational power. In my second thesis group, I will introduce the virtual topographic processor arrays, which enable to trade the resolution to speed.

To be able to efficiently design and use multi/many core architectures, we have to answer the major questions, namely which processor arrangement to use in a particular application, and how to implement the algorithm on them. In my third thesis group, I am classifying the wave type operators according to their implementation methods on the different architectures. Then, I am introducing an architecture selection method.

Thesis I Direct CNN template design

Cellular Neural/nonlinear Networks (CNNs) implement coupled, nonlinear, differential equation systems, which are continuous in time and value, and discrete in space. The template of the CNN lists the free parameters of the network, which is practically the program of this complex array processor. Based on the template value configuration, the region of dependency can be either within the interconnection radius; or within a well defined bounded region, which is larger than the interconnection radius; or it can be the whole CNN lattice without any limitations. In the latter case, the transients of the coupled differential equation system may exhibit spatial-temporal wave propagation phenomena.

Finding appropriate template to solve a certain 2D function is an inverse problem, because it is easy to test, what function belongs to a template, but it is not trivial the other way around. There are various methods for finding these templates. The first is the heuristic, which may or may not lead to some solutions, which are usually not optimal. The second is the template learning with different methods (back propagation [34], genetic algorithms [33], gradient based method [34], etc). These methods often require long iteration sequences, and lead to sub-optimal solutions only, if they find solution at all. Since the template space is extremely huge, due to the 19 free variables for the simplest CNN, the brute-force method is not an option. The last method is the direct template design, which leads to optimal solution by using a few closed forms. I have derived these forms for binary-input \rightarrow binary-output templates.

- I. I have developed a method for directly designing optimal binary-input \rightarrow binary-output CNN templates for both non-propagating and propagating cases [1]. The method reduced the inverse problem of the CNN template design to the solving of

a set of inequalities. The advantage of the method is that it leads to the optimal (most robust) template.

- I.1. I have proved that in case of binary-input \rightarrow binary-output propagating type templates, the pixel transition is strictly monotonic if the off-center **A** template elements are positive, and $a_{00} > 1$, and mono-directional cell transitions are enabled only.

I have introduced global rules, which were derived from the properties of the particular wave. These global rules were then transformed to local rules, and later to activation patterns. These activation patterns were template sized binary or symbolic patterns, which defined those local binary pixel arrangements where the output was supposed to change. I showed that even propagating case wave phenomena can be described using such activation patterns. I have classified the binary-input \rightarrow binary-output templates, and showed how to generate template forms, and then systems of inequalities from the activation patterns.

Thesis II Virtual topographic processor arrays for space to time conversion

Topographic array processors, such as classic CNN computer chips, can efficiently utilize thousands of parallel processors, and can even process medium sized images above 10,000 FPS easily using less than 1 Watt. Unfortunately, this extremely high image processing performance and efficiency cannot be traded to higher resolution–lower speed operation mode due to inherent architectural constraints of the topographic arrangement.

The problem is rooted in the fact that the data is distributed in the 2D topographic array among the processor cells and each processor cell handles one pixel (fine-grain), or a small image segment (coarse-grain). This requires keeping the entire image in internal local memory, otherwise each of the processor cells needs parallel access to external memories during the operation, which is impossible considering couple of hundreds or thousands processors on a single chip.

I bridged the problem by showing different virtual high resolution topographic arrays. The physical processor engine, behind the virtual processor array is a medium resolution topographic processor array. This physical processor array is allocated to different parts of the virtual processor array from time to time. The allocation can be either periodical or non-periodical. In the latter case, it is image content dependent. The topology of the physical processor array and its communication methods depend on the approach and the parameters such as resolution, pixel clock, signal domain, neighborhood size, and functionality. Beyond the general idea, I have explored four particular arrangements.

- II. I have described the virtual topographic processor array concept. Two approaches have been introduced for the time-to-space conversion of the topographic computer arrays. The first is a sequential approach for early image processing, while the second is the foveal approach for post processing.
- II.1.1 I have introduced a virtual processor array architecture for analog video image processing. The special feature of the architecture is that it does not digitize the analog video signals. Behind the 2D full video frame size virtual processor array, there is an elongated mixed-signal physical processor array [4].
- II.1.2 I have proposed a virtual processor array architecture for calculating high speed CNN operation in the digital domain. The physical processor array is the CASTLE [2] architecture, which can achieve calculations in 3 different bit depths. The virtual processor array can handle various image sizes, and can implement space variant template operations.
- II.2.1 I have showed how the foveal concept can be implemented by using ultra-high speed, low resolution CNN chip as physical device. This virtual processor approach bridged the gap between the 10,000 FPS low resolution image processing, and the video speed high-resolution image processing. With the help of the ultra-high speed, low resolution CNN chip, 1000 FPS foveal image processing was performed in high-resolution images.
- II.2.2. Monolithic, ultra-low power, ultra-high performance sensor-processor chip is proposed for performing airborne navigation tasks. The novelty of the device is that it combines a medium resolution mixed-signal processor array for early image processing, and a digital foveal processor array for post processing. The single chip implementation was made possible by using the new 3D silicon integration technology.

Thesis III Low-power processor array design strategy for solving 2D topographic problems

Low-power topographic processor arrays are widely used in the image processing field nowadays. Their characterization and comparison is an important task, when someone wants to select one of them to apply in a particular application. However, it is quite complicated since they have drastically different architectures, operation modes, and parameters. To be able to make this comparison, I have analyzed the most important low-power topographic and non-topographic multi- and many-core architectures, and then, I have classified the basic image processing operators, including the wave computing ones, according to their

implementation methods on these architectures. Based on these results, I gave an optimal multi- or many-core architecture selection methodology.

- III.1. I have classified the most important 2D operators, including the wave computing ones, based on their implementation methods on different topographic and non-topographic multi- and many-core image processing architectures. The most distinguishing features were the activity pattern distribution of the pixels (front active versus area active), the content dependency of the waves, and the spatial-temporal calculation method of the operators.
- III.2. I have determined the computational efficiency of these 2D operators, on various topographic and non-topographic multi- and many-core image processing architectures. Moreover, I have calculated the computational demand, the execution time, and the latency values of the operators executed on the different topographic architectures.
- III.3. I have derived an optimal multi- or many-core image processing architecture selection methodology. The method is based on the characteristic features of the given algorithms, namely the frame-rate, latency, resolution, instruction types, and the structure of its flow-graph.

2 Direct CNN template design

A Cellular Neural Network (CNN) [26] is a locally interconnected analog processor array arranged to regular 2D grid. Its two-dimensional inputs and output make it extremely suitable for image processing. Due to its regular (in most cases rectangular) arrangement and its local interactions, programmable CNN (the so-called CNN Universal Machine [27]) can be efficiently implemented on silicon. With the today available deep-submicron technology 128×128 sized analog processor arrays have been implemented on a single chip [42]. The spatial-temporal transient of an analog VLSI CNN array settles down in the microsecond range. This means that an image processing primitive (like edge detection, blurring, sharpening, thresholding, etc.) can be calculated for a 128×128 pixel sized image in a few microseconds on a single low-power chip. This speed enables to implement image processing algorithms and real-time visual decisions over 10,000 FPS in the embedded space, which is unique. But while programming conventional digital computers is relatively easy, here one has to find an appropriate parameter set of a continuous time spatial-temporal nonlinear processor array to force its dynamics to calculate an image processing primitive. Since CNN has space invariant local interconnection structure it has a few dozens free parameters only, depending on the sphere of influence. This parameter set, called *template*, exclusively determines its array dynamic behavior.

There are three major template design methods: (i) intuitive, (ii) template learning, and (iii) direct template design. The first requires intuitive thinking of the designer. In some simple cases it leads to quick results, but typically not to the optimal one. On the other hand, it does not guarantee to find the desired template at all. Moreover, designers need to have lots of experiments in both the image processing and the array dynamics.

The second design method, the template learning, is an extensively studied, popular field of the CNN research. Almost all classic neural network training methods have been adapted to the CNN structure. But there are three serious problems with these techniques and results. First of all, the learning is based on input and desired output pairs and during the learning procedure better and better results are supposed to be generated with better and better templates. But in many cases (especially binary propagating type templates like CCD shown in Figure 16 [23]) a template either works or does not work, and there are no gradual better and better result series. This changes the strategy of a learning method to a brute-force one. The second problem is that in some cases the template for the given problem does not exist. The learning methods cannot even realize it and run forever. If the template exists, it might take a long time to find it, if it can at all. The third problem is that several proposed learning methods were tested and proved to be efficient to those template classes, which can be directly derived from the exact function descriptions. In these cases it is unnecessary to use learning methods. However, there are some cases when the template learning is a very

important design method. In these cases, no explicit desired output exists, hence direct template design is impossible. Texture separation is a good example for this case [51].

The third method, what I have introduced, is the direct template design. It can be applied when the desired function is exactly specified. The design methods depend on the particular template class. While the previous two methods produce a single or a few operational templates, here we get all of them. This provides the opportunity to choose the most robust one among them. Moreover, this method needs only a small fraction of the computational power with respect to the template learning needs.

All template design methods, presented in this paper, can be used in both the *Chua-Yang model* [24] and the *Full Signal Range (FSR) model* [36]. In case of the FSR model the center element of the derived template should be decreased by 1.

In this chapter, first a brief description of the CNN will be given. This will be followed by the uncoupled binary-input \rightarrow binary-output CNN template design. Finally the coupled binary-input \rightarrow binary-output CNN template design closes the chapter.

2.1 The Cellular Nonlinear/Neural Network model

A CNN is defined by the following principles:

- A set of spatially coupled dynamical cells arranged to a 2D grid, where information can be loaded into each cell via two independent variables called input (u) and the initial state ($x(0)$). In this way, the CNN is continuous in time and value, and discrete in space.
- The behavior of the network is defined by the coupling law which describes the relation (weight coefficient) between one or more relevant variables of each cell to all local neighboring cells located within a prescribed sphere of influence $S_r(i,j)$ of radius r centered at cell (i,j) . This set of coupling laws, called template, is space invariant for the whole grid.

Figure 1a shows a CNN composed of cells that are connected to their nearest neighbors. Due to its symmetry, regular structure and simplicity this type of arrangement (rectangular grid) is primarily considered in all implementations.

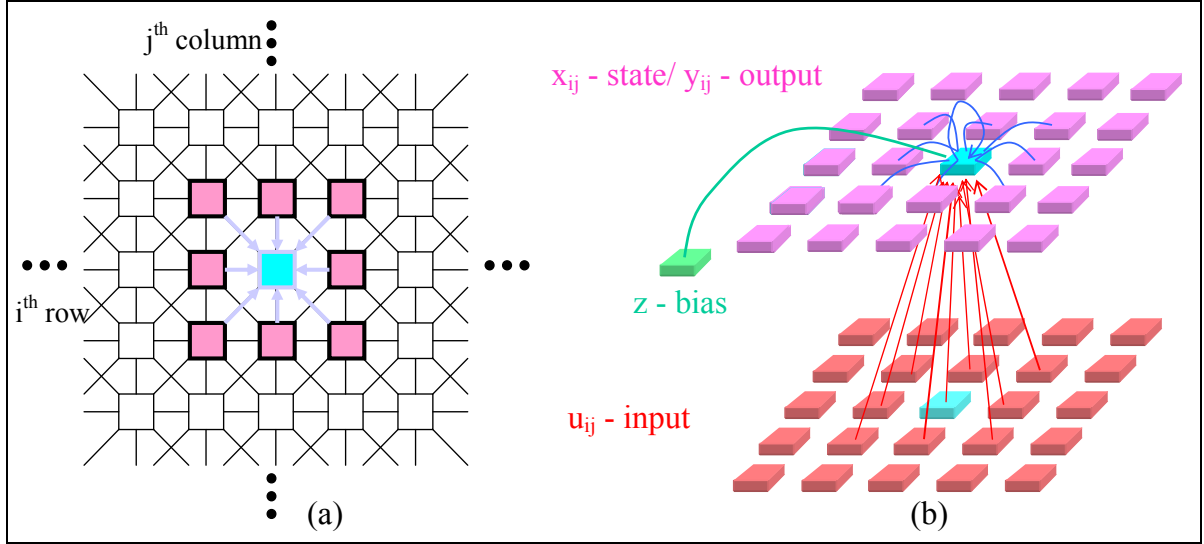


Figure 1. A 2-dimensional CNN defined on a square grid (a). The i,j -th cell of the array is cyan, whereas cells that fall within the sphere of influence of neighborhood radius $r = 1$ (the nearest neighbors) are pink. The two layers of the CNN and their internal interconnections are shown in (b). The red arrows represents the feed forward weights, while the blue ones the feed-back. The green arrow shows the space invariant bias.

Figure 1b shows the two layers of the CNN. The lower layer is the input, which is typically constant in time. The upper one represents the dynamically changing state and output layers. They are indicated as a single layer because they are strongly bonded (see equation 2.2). The external single source is a constant bias. The electrical circuit of the cells is shown in Figure 2. The cell dynamics is described by the following nonlinear ordinary differential equations:

State equation:

$$\dot{x}_{ij}(t) = -\frac{1}{\tau}x_{ij}(t) + z_{ij} + \sum_{k,l \in S_r(i,j)} A_{ij;kl} \cdot y_{kl}(t) + \sum_{k,l \in S_r(i,j)} B_{ij;kl} \cdot u_{kl}(t) \quad (2.1)$$

Output equation:

$$y_{ij}(t) = f(x_{ij}(t)) = \begin{cases} 1 & \text{if } x_{ij}(t) > 1 \\ x_{ij}(t) & \text{if } -1 \leq x_{ij}(t) \leq 1 \\ -1 & \text{if } x_{ij}(t) < -1 \end{cases} \quad (2.2)$$

The graph shows the output function y_{ij} versus the state variable x_{ij} . The function is a saturation function, where $y_{ij} = 1$ for $x_{ij} > 1$, $y_{ij} = x_{ij}$ for $-1 \leq x_{ij} \leq 1$, and $y_{ij} = -1$ for $x_{ij} < -1$. The x-axis is labeled x_{ij} and the y-axis is labeled y_{ij} .

where

- x_{ij} , y_{ij} , u_{ij} are the state, the output, and the input variables of the specified CNN cell, respectively. The state and output vary in time, the input is static (time independent), ij refers to a grid point associated with a cell on the 2D grid, and $kl \in S_r$ is a grid point in the neighborhood within the radius r of the cell (i,j) .
- z_{ij} is the threshold (also referred to as bias) which is constant in space and time.

- Term $A_{ij,kl}$ represents the feedback, $B_{ij,kl}$ the control weight coefficients. These are scalar matrices, which are constant in space and time during a transient.
- τ is the cell time constant, for simplicity we consider $\tau=1/RC=1$.
- Function $f(\cdot)$ is the output nonlinearity, in our case a unity gain sigmoid. It has three regions. The middle region is called linear region, while the two other regions are called saturation regions. The CNN terminology calls an output value grayscale, when it is in the linear region, and binary value, when it is in one of the saturation regions.
- t is the continuous time variable.

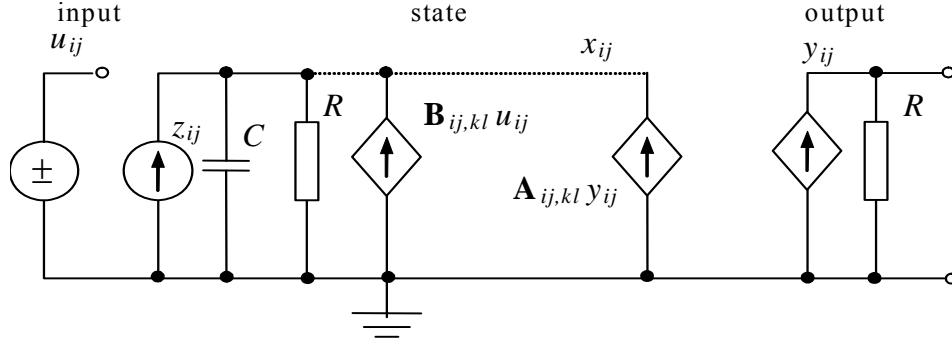


Figure 2. The equivalent circuitry corresponding to the CNN state and output equations as it is defined in (2.1). The control and feedback terms are represented by voltage controlled current sources ($B_{ij,kl}$ and $A_{ij,kl}$).

The state equation (2.1) and the output equation (2.2) define a coupled nonlinear differential equation set, which is a rather complex framework for computation. The first part of the state equation is called cell dynamics, whereas the additive terms following it represents the synaptic interactions.

The time constant of a CNN cell is determined by the linear capacitor (C) and the linear resistor (R) and it can be expressed as $\tau = RC$. A CNN cloning template, which can be considered as the instruction on the CNN array, is given by two weight coefficient matrices and a bias term (for example see equation 2.3) implemented by the voltage controlled current sources.

$$\mathbf{A} = \begin{bmatrix} a_{-1,-1} & a_{-1,0} & a_{-1,1} \\ a_{0,-1} & a_{0,0} & a_{0,1} \\ a_{1,-1} & a_{1,0} & a_{1,1} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{-1,-1} & b_{-1,0} & b_{-1,1} \\ b_{0,-1} & b_{0,0} & b_{0,1} \\ b_{1,-1} & b_{1,0} & b_{1,1} \end{bmatrix}, \quad z = i; \quad (2.3)$$

In order to specify fully the dynamics of the array, the boundary conditions have to be defined. Cells along the edges of the array may see the value of cells on the opposite side of the array (toroidal boundary), a fixed value (Dirichlet-boundary) or the value of mirrored cells (zero-flux boundary).

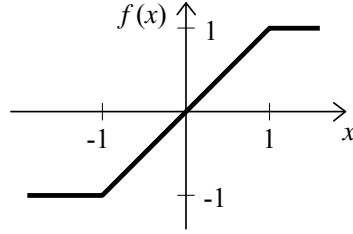
2.2 Uncoupled CNN templates

Uncoupled templates have zero off-center \mathbf{A} template elements only. The general form of the uncoupled templates is the following:

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & a_{00} & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{-1-1} & b_{-10} & b_{-11} \\ b_{0-1} & b_{00} & b_{01} \\ b_{1-1} & b_{10} & b_{11} \end{bmatrix}, \quad z = i \quad (2.4)$$

Since their dynamic parts are uncoupled, they work as an array of independent first-order elements (cells). Hence, it is satisfactory to analyze the dynamic behavior of a single cell only. A single cell receives 9 static inputs from the input layer (u_{kl}). It has an initial condition $x(0)$, which can be considered as a tenth input. An uncoupled CNN cell maps these 10 inputs to a single output, with other words it implements a 10 input one output function. The state equation of a single cell is as follows:

$$\begin{aligned} \dot{x}(t) &= -x(t) + a_{00}y(t) + s \\ s &= \sum_{C(kl) \in N_r(i,j)} \mathbf{B}_{ij,kl} u_{kl} + z; \\ y &= f(x) \end{aligned} \quad (2.5)$$



Where: s is a constant during the transient evaluation, which depends on the template and the input of the CNN; f is the output characteristics, a sigmoid function in our case. The flow-graph which describes the first order, differential system of a single cell can be seen in Figure 3.

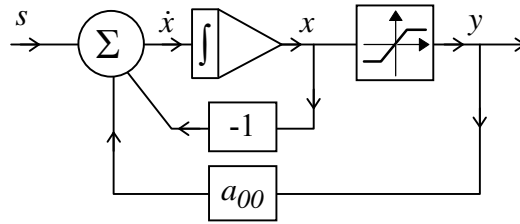


Figure 3. The flow-graph of the first order system of a single cell.

Before going on with the dynamic analysis of a single cell, let us describe the errors and deviations coming from the analog implementation of the CNN. In case of the analog implementation, the template parameters of the CNN are slightly varying. This can be considered as each cell has an individual template. Although all the template values are close to the nominal ones the difference of any measurable template value and the corresponding nominal one changes from value to value and cell to cell, but they are constant in time. Their distribution and deviation depends on the particular chip piece.

The dynamics of a cell depends on the a_{00} parameter. First let us see the dynamic response of the cell quantitatively, via solving the ODE in the linear region ($x=y$).

$$\dot{x} + (1-a)x = s \quad (2.6)$$

$$\text{if } a \neq 1: \quad x(t) = \frac{s}{1-a} + ce^{-(1-a)t} \quad (2.7)$$

$$\text{if } a = 1: \quad x(t) = \int s dt + c = st + c \quad (2.8)$$

After the quantitative analysis, let us do a qualitative analysis of the practically important cases. We call the following statements *rules*, and we will recall them later on. These are as follows:

Rule 1. $a_{00}=0$. In this case the final state of the cell is equal to the input ($x_{\infty}=s$), hence the final output is $y_{\infty}=f(s)$.

- Notes:
1. The final output is independent of the initial state $x(0)$.
 2. The transient settles with an exponential decay in the linear region.
 3. The output will be binary if and only if $|s|>1$.

Rule 2. $a_{00}=1$. In this case the CNN behaves as an integrator, while x is in the linear region ($|x|<1$). Ideally, if $s \neq 0$, then the final output depends on the sign of s , i.e.: $y_{\infty}=\text{sign}(s)$. If $s=0$, then the final output depends on the initial state, i.e.: $y_{\infty}=x(0)$.

However, due to the given mismatch and other non-idealities of the analog VLSI implementation of the circuit, we have to use $|s|>\epsilon>0$, to get $y_{\infty}=\text{sign}(s)$. (ϵ is larger than the largest template mismatch value)

- Notes:
1. If $|s|>\epsilon>0$, then the final output is independent of the initial state $x(0)$, and it is binary.
 2. If $|s|>\epsilon>0$, then the output saturates in less than $2s$ time (starting from the linear region). In this case the transient decay is linear in the linear region.
 3. In practice, we have to avoid the use of the $s=0$ case, because in case of an analog realization the precise equality is not a valid possibility, hence the final output will depend on unpredictable parameter deviations.

Rule 3. $a_{00}>1$, usually 2 or even higher. The final output is always binary, due to the positive feedback loop. The final output depends on the initial state and the

contribution of the input (s). There are three practically important cases. The last discussed case is the general case.

- (a) $x(0)=0$ hence $y(0)=0$. The final output depends on the sign of s , i.e.: $y_{\infty}=\text{sign}(s)$. In this case the initial feedback is zero. The integrator starts increasing or decreasing according to the sign of s , and due to the positive feedback, it will go to one of the saturation zones.
- (b) $x(0)=+1$ hence $y(0)=+1$. The final output remains $+1$, if $a_{00}+s-1>0$. The final output changes to -1 , if $a_{00}+s-1<0$.
- (c) $x(0)=-1$ hence $y(0)=-1$. The final output remains -1 , if $-a_{00}+s+1<0$. The final output changes to -1 , if $-a_{00}+s+1>0$.
- (d) $x(0)=x_0$, where $|x_0|<1$, hence $y(0)=x_0$. The final output depends on the sign of $\dot{x}(0)$, precisely: $y_{\infty}=\text{sign}(\dot{x}(0))=\text{sign}((a_{00}-1)x_0+s)$.
(This case is the generalization of the previous ones.)

Note: The $a_{00}+s=1$ situation and the $-a_{00}+s=-1$ situation should be avoided in practical cases, because in case of an analog implementation the precise equality is not a valid possibility, hence the final output will depend on unpredictable parameter deviations.

These rules can be trivially derived from the state equation (2.5) and the flow-graph (Figure 3.) of a single CNN cell. After analyzing the dynamic of a single cell, here we go through the elementary uncoupled CNN template classes. From now the inputs and the outputs of the CNN will be presented in image forms. We follow the original convention of [24], where $+1$ stands for black, and -1 stands for white, and the intermediate values are represented by different gray shades.

2.2.1 Binary-input \rightarrow binary-output uncoupled CNN templates

The uncoupled binary CNN templates form a very important class of the CNN templates, because they cover many different frequently used image processing tools, including the binary mathematical morphology. The family of operations can be separated into two main groups. The first is the single input (e.g. constant zero initial state, image on input layer only), while the second is the two input (images on both the initial state and the input layers).

During the discussions of the template design methods, first we introduce the design for some special template classes, and then the general solution will be discussed. These template classes are important, because they are simpler than the general solution, hence the template design methods are simpler too, and moreover they cover most of the practical templates. We will also give the list of the templates from the Template Library [23] belonging to the each design classes.

Class I. Single input image, equal pixel roles

This class of templates extracts 3×3 unweighted pattern combinations. (Unweighted means that the individual elements play the same role). When describing the problem a binary 3×3 pattern and a limit (integer number) are given. The binary pattern contains black pixels, white pixels and “don’t care” pixels (See example in Figure 4.). The given limit controls that at least how many positions of the pattern should match to set a pixel.

The black-and-white input image is placed to the input of the network. The initial state is set to zero. At the end of the operation, the output is black in those pixel positions, where the number of matches was equivalent or exceeded the given limit.

Design example A:

Given the 3×3 binary pattern shown in Figure 4a. Suppose that the threshold value is 5.

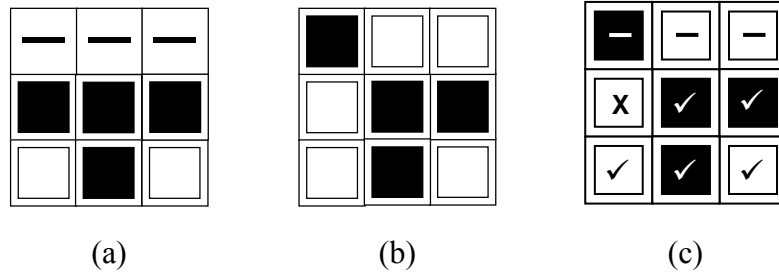


Figure 4. Example for binary pattern matching. (a) shows the binary pattern. Squares with ‘-’ means, “don’t care”. (b) shows the test pattern. (c) shows the matching and the non-matching pixel locations. The matching positions are denoted with ‘✓’ and the non-matching one with ‘X’. Since, there are 5 matching positions, the output of the cell will be black (+1).

The design steps of the uncoupled CNN templates are shown by the flowchart in Figure 5. The first step is the most important, because the key of the successful template design is the correct template form determination. As we saw in (2.4), generally there are 11 free parameters of the uncoupled CNN templates. When we determine the template form, we drastically reduce the number of the free parameters. Some of the parameters will be set to zero, and some groups of it will be handled together. With this method the number of the free parameters is usually reduced to 3 or 4. This means that in usual cases the template space is reduced to a 3 or 4 dimensional one. See (2.9) for the template form of the design example 1!

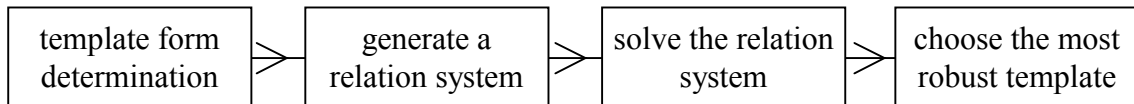


Figure 5. The flowchart of the design method of the binary input-binary output templates.

The second step of the template design is the generation of a system of inequalities. It can be derived automatically from the task and the *Rules*. Each inequality guarantees the output to a certain input configuration. Since the input-output pairs are known, the generation of the system of inequalities is simple. Each relation defines a hyper plane, which cuts reduced template space into two halves. The inequality is satisfied in one half only. Since all the inequalities should be satisfied, the intersection of the half spaces contains the correct templates. If it is an empty set, the function cannot be solved with a single template (linearly not separable function [29]) in the determined template form. A graphical visualization example can be seen in Figure 6a, and will be explained in the next example.

Template form determination:

After the general idea of the design method was explained (Figure 5) let us show it in practice in Example I. First of all, the template form should be determined. The template form can be directly derived from the binary pattern (Figure 4a). a_{00} will be larger than 1 (say 2) which guarantee that the final output will be binary (*Rule 3.*). The initial condition will be set to zero, hence the final output will be $sign(s)$ (*Rule 3a.*). In template **B**, the don't care positions are equal to zero. All the black positions play the same role, hence, they can be denoted with the same free parameter, say b . The role of the white positions are exactly the opposite of the role of the black positions, hence they will denoted by $-b$. The template is sought in the following form:

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ b & b & b \\ -b & b & -b \end{bmatrix}, \quad z = i \quad (2.9)$$

System of inequalities:

After determining the form of the template the generation of the system of inequalities is straightforward. One has to go through all the possible combinations of the input patterns, and apply the particular Rule, in our case *Rule 3a*. This means that the initial state is zero, and the $sign(s)$ determine the final output. Numerically we can distinguish 7 different cases depending on the number of the matching pixels.

<i># of matching pixels</i>	<i>desired output</i>	<i>relations</i>	(2.10)
6	black (+1)	$6b+i>0$	
5	black (+1)	$4b+i>0$	
4	white (-1)	$2b+i<0$	
3	white (-1)	$i<0$	
2	white (-1)	$-2b+i<0$	
1	white (-1)	$-4b+i<0$	
0	white (-1)	$-6b+i<0$	

Solution of the system of inequalities, and selection of the most robust template:

Fortunately in this case there are only two free parameters of the system, hence we can solve the problem graphically. The graphical solution can be seen in Figure 6a. By solving the system of inequalities we get an infinitely large subspace, from which we have to pick a single point to be the nominal template. By testing different templates from the found region in simulator, we can see that the convergence of some templates will be faster, others will be slower, but all templates in the specified template sub-space will work fine. But if we want to apply our templates on a CNN chip we have to consider the parameter deviations coming from the analog implementation. As we saw at the beginning of this section, the parameter deviation can be considered as each cell would have an individual template, which is close to the nominal template. To select the most robust template, we have to consider the followings:

- It is a rule of thumb that the more we scale up the template values, the faster the transient will be.
- Due to local silicon process variants, we suppose that the template values in a CMOS chip will be within a circle around the nominal template. To guarantee the robustness of the template this circle should be inside the specified subspace with its total volume. On the other hand, it can be seen that the subspace opens (becomes wider) if the values are scaled up.
- The analog implementation of the CNN always limits the maximal absolute value of the template elements. Let say that in our case the absolute value of a template element should not exceed 3 and the absolute value of the bias (current) should not exceed 6. It is the case in [42]. This limits the infinite subspace to a finite subspace. These boundaries are denoted with dashed lines in Figure 6b.

Hence, we have to choose the largest possible b and i value from the middle of the subspace. These values ($b=2.2$, $i=-6$) determine the selected template. We call the selected template as the nominal template. The real templates will be around it in the circle. The chosen best template is the following:

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 2.2 & 2.2 & 2.2 \\ -2.2 & 2.2 & -2.2 \end{bmatrix}, \quad z = -6 \quad (2.11)$$

Notes:

1. The specialty of this template class is that template \mathbf{B} contains one free parameter only, hence it is constructed from zero, a certain real number and its opposite.

2. From the robustness point of view, it is more difficult to implement the template, if the threshold number is larger (6 instead of 5 in our case), because it makes the result template subspace narrower. In Figure 6a, this subspace is the narrow one below the shaded part. If the resulting template subspace is narrower, it might be difficult to keep the circle of tolerance with its total area inside.

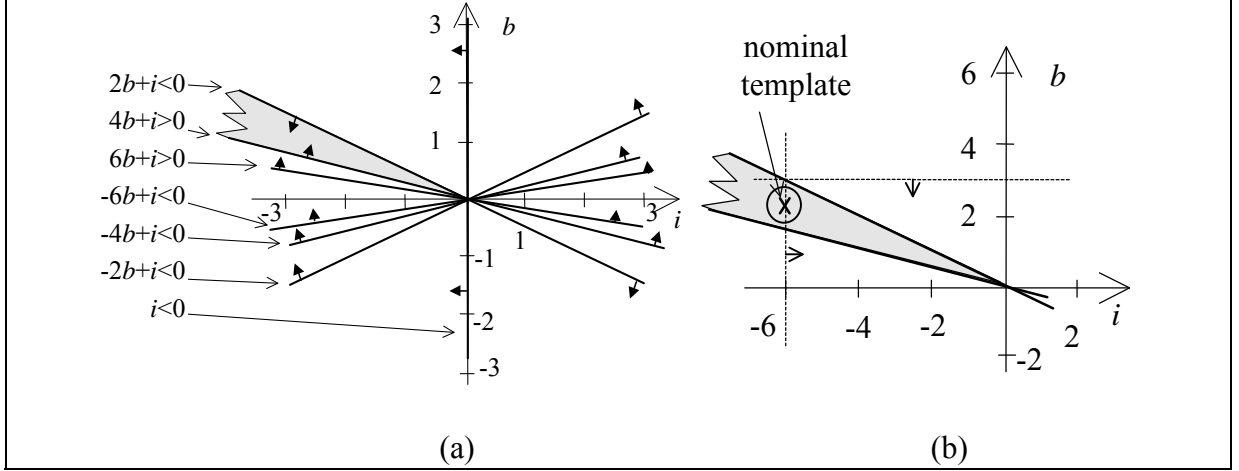


Figure 6. (a): Graphical solution of the system of inequalities (2.10). All of the inequalities are represented with a straight line, which divides the plane to two halves. The arrows on each line indicate that half, which satisfies the particular inequality. The union of the half plans is the solution subspace (shaded). (b): Selection of the nominal template. The dashed lines show the technical limitations of the ACE16k chip. The 'x' shows the chosen best nominal template, and the circle around it contains the real templates.

Templates from the Template Library [23], which belong to this class:

EROSION, DILATION, DELVERT1, DIAG1LIU, FIGDEL, LSE, PEELHOR, RIGHTCON. (Some of these templates are described in the Appendix.)

Class II. Design method of the one input image differential pixel roles

In the previous case, all the pixels played the same role, and the decision was made on their matching statistics. Here, we have two groups of active pixels. The first pixel group contains the priority pixels, which must match anyway, while the second group contains the non-priority pixels, from which only a given number is required to match. In this template class, a binary pattern (with indicated priority, non-priority, and don't care positions), a threshold (limit) number, and a rule whether to change white pixels to black or black pixels to white are given. When the number of the matching positions is calculated the non-priority positions should be concerned only.

Design example B:

Given the 3×3 binary pattern shown in Figure 7a. The task is to set those locations to white, where the priority pixel and all the five non-priority pixels matches, and keep the original value otherwise. Figure 7b and c show an example. (The example template is the first one from the skeletonization template series [23].)

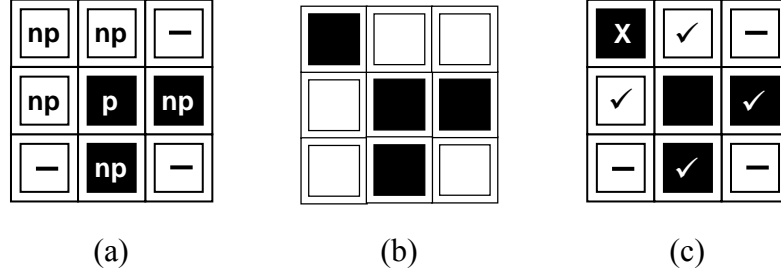


Figure 7. (a) is the given binary pattern with the indicated priority (p), non-priority (np) and don't care (-) positions. (b) is the test pattern. (c) shows the matching and the non-matching pixel locations. Since, there are 4 matching positions in the non-priority region the output of the cell will not change. Note that when the matching positions are calculated the priority pixel position is not concerned.

Template form determination:

The template form can be directly derived from the binary pattern (Figure 7a). a_{00} will be larger than 1 which guarantee that the final output will be binary (*Rule 3.*). The initial state will be zero, hence the final output will be determined by *Rule 3a* (the sign of s). In template **B**, the don't care positions are equal to zero. The specialty of this class is that the priority positions of template B play different role than the non-priority positions. The reason is that all of the priority ones are supposed to match. Hence, the priority pixel positions of template B always get a new free parameter, (say b).

The black non-priority positions play equivalent roles, hence they can be characterized by the same free parameter, name it c . The role of the white non-priority positions play exactly the opposite role than the black positions, hence they will be $-c$. The template is sought in the following form:

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} -c & -c & 0 \\ -c & b & c \\ 0 & c & 0 \end{bmatrix}, \quad z = i \quad (2.12)$$

System of inequalities:

Since the initial state of the CNN is zero here and $a_{00} > 1$, we have to consider *Rule 3a*. Here the number of the relations will be $(N_p + 1) * (N_{np} + 1)$, where N_p and N_{np} are the number of the priority and non-priority positions respectively. In our example, the inequalities are as follows:

<i>self input</i> (<i>priority</i>)	<i># of matching non- priority pixels</i>	<i>desired output</i>	<i>relation</i>	(2.13)
black (+1)	5	white (-1)	$b+5c+i-1<0$	
black (+1)	4	black (+1)	$b+3c+i-1>0$	
black (+1)	3	black (+1)	$b+c+i-1>0$	
black (+1)	2	black (+1)	$b-c+i-1>0$	
black (+1)	1	black (+1)	$b-3c+i-1>0$	
black (+1)	0	black (+1)	$b-5c+i-1>0$	
white (-1)	5	white (-1)	$-b+5c+i+1<0$	
white (-1)	4	white (-1)	$-b+3c+i+1<0$	
white (-1)	3	white (-1)	$-b+c+i+1<0$	
white (-1)	2	white (-1)	$-b-c+i+1<0$	
white (-1)	1	white (-1)	$-b-3c+i+1<0$	
white (-1)	0	white (-1)	$-b-5c+i+1<0$	

After solving the system of inequalities, the resulting template is as follows:

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0.5 & 2.5 & -0.5 \\ 0 & -0.5 & 0 \end{bmatrix}, \quad z = -0.5 \quad (2.14)$$

Templates from the Template Library [23], which belong to this class:

CORNER, EDGE, SKELETONIZING, CENTER, FIGEXTR, JUNCTION, CONCAVE

Class III. Two input images

The specialty of this class is that both the input and the initial state of the CNN carries two different relevant images, hence an additional input appears, and the total number of the pixels, which affects the final output is 10 (instead of 9 like in the previous two classes). The image downloaded to the initial state of the network can be considered as a mask. This means that we cannot define a neighborhood operation on the initial state. Rather than that, through this image, we can modify the local neighborhood functionality applied to the other image downloaded to the input.

On the other image, downloaded to the input, the same spatial functions can be defined what we saw in the previous two classes. The resulting image of this function and the initial state can be logically combined with the same template.

The general solution of this class is as follows. If we consider Rule 3b and c, we find that the final output is +1 if:

$$w_{00} + s > 0 \quad \text{if } x(0) = +1 \quad (2.15)$$

$$-w_{00} + s > 0 \quad \text{if } x(0) = -1 \quad (2.16)$$

Here we used $a_{00} = 1 + w_{00}$, because the '1' is used for the compensation of the integrator in the linear region, and the remaining w_{00} is the real weight coefficient. Similarly, the final output is -1, if:

$$w_{00} + s < 0 \quad \text{if } x(0) = +1 \quad (2.17)$$

$$-w_{00} + s < 0 \quad \text{if } x(0) = -1 \quad (2.18)$$

The consequences of the above expressions are:

$$y_{\infty} = -1 \quad \text{if } s < -w_{00}$$

$$y_{\infty} = 1 \quad \text{if } s > w_{00} \quad (2.19)$$

$$y_{\infty} = x(0) \quad \text{if } -w_{00} < s < w_{00}$$

This leads to a hysteresis behavior, as it is shown in Figure 8. The output depends on the logic combination of the contributions of the input and the initial state. Three kinds of logic combinations are possible:

- AND, if $s < w_{00}$
- OR, if $s > -w_{00}$
- The third one is a non-standard logic. In this case s can extend the range of $[-w_{00}, w_{00}]$ in both directions. The output will be defined by the contribution of the input in that cases when $|s| > |w_{00}|$, otherwise it will be $x(0)$.

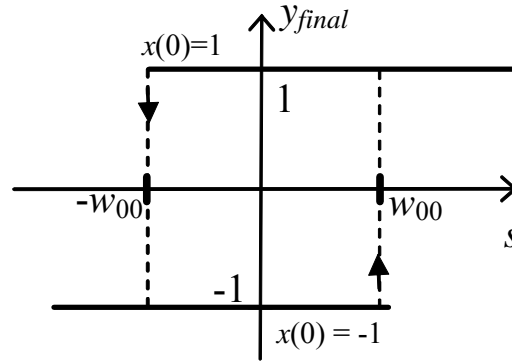


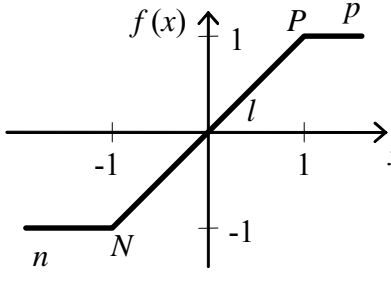
Figure 8. Hysteresis phenomenon can be found in the final output of the binary-input \rightarrow binary-output, two-input, uncoupled CNNs when the self feedback is larger than 1.

Templates from the Template Library [23], which belong to this class:

LOGAND, LOGDIF, LOGOR, LOGORN.

2.3 Coupled CNN templates

A CNN with coupled template has array dynamics. The change of the output of an individual cell effects its neighbor's output and vice versa. The array dynamics is described by the coupled first order differential equation system [24] shown in (2.20). The output characteristic, called sigmoid function, is also shown in (2.20). The u , l , and p letters denotes the negative saturation, the linear, and the positive saturation regions respectively, while the N and the P indicates the negative and the positive break points.

$$\begin{aligned} \dot{x}(t) &= -x(t) + \sum_{C(kl) \in N_r(i,j)} \mathbf{A}_{ij,kl} y_{kl}(t) + s \\ s &= \sum_{C(kl) \in N_r(i,j)} \mathbf{B}_{ij,kl} u_{kl} + z; \\ y &= f(x) \end{aligned} \quad (2.20)$$


We separated the contribution of template \mathbf{B} and the bias term, because they are constant during the spatial-temporal dynamics. In this section, we suppose that $a_{00} > 1$.

A coupled Cellular Neural Network structure allows propagation phenomenon. The propagation always works in such a way that only some cells are active and the rest are inactive, in the array. The active cells may or may not activate their inactive neighbors, and after a while they became inactive again. The activated neighbors may or may not activate new neighbors, and the wave propagates as long as active cells can find new neighbors to activate.

A cell is considered to be inactive in a certain time instant, if it is in a stable equilibrium point. The state value of an inactive cell must be in the saturation region (n or p), because due to the positive self-feedback ($a_{00} > 1$), the cell cannot be in a stable equilibrium point in the linear region [24]. Rule 4 describes the necessary conditions for a cell to be stable in the saturation region.

Rule 4. As it follows from (2.20) a cell is stable in the positive saturation region (p) if the value of the $\sum_{C(kl) \in N_r(i,j)} \mathbf{A}_{ij,kl} y_{kl}(t) + s$ term is larger than +1, or it is stable in the negative saturation region (n), if this value is smaller than -1.

A cell is considered to be active in a certain time instant, if its output is changing. The state of an active cell is always in the linear region. Rule 5 shows the necessary condition to activate a cell.

Rule 5. A cell (pixel) leaves a saturation region under the following conditions:

- (a) a cell leaves the positive saturation region
(moves from p to l by crossing P),
if:

$$\sum_{C(kl) \in N_r(i,j)} \mathbf{A}_{ij,kl} y_{kl}(t) + s < 1; \quad (2.21)$$

- (b) a cell leaves the negative saturation region
(moves from n to l by crossing N),
if:

$$\sum_{C(kl) \in N_r(i,j)} \mathbf{A}_{ij,kl} y_{kl}(t) + s > -1. \quad (2.22)$$

In most cases, an activated cell migrates from one saturation region to the other, typically on a monotonic way. Certainly, in some propagating waves some pixel arrangements produce situations, in which cells go into the linear region from one saturation region, and after a while, it changes course and go back to the same saturation region, where it was coming from. However, in these cases, the final output cannot be unambiguously derived from the input in real analog CNN implementations, because the fact, whether a cell changes course or not, may depend on the local noise of the analog cell (See Section 2.3.1.2). Therefore, it is better to use waves, where such situations are excluded.

We can distinguish propagating waves (template configurations), which enable only one directional (mono-directional) cell transition, and those, which enable both. Though the proposed method can generate templates for both kinds of propagating waves, it is better to design templates for one directional cell transitions, because in that case, one can make sure that the cells will monotonically cross the linear region, making the output an unambiguous function of the input. Rule 6 shows the necessary conditions, which allow mono-direction cell transition only. The specialty of Rule 6 is, that it defines this property by using template values purely.

Rule 6. A cell is stable in the positive saturation region (p) if the derivative of its state cannot be negative in the positive break point (P):

$$\dot{x}_{ij}(t) = w_{00} y_{ij}(t) + \sum_{ij,kl}^{kl \neq 00} \mathbf{A}_{ij,kl} y_{ij,kl}(t) + \sum_{ij,kl} \mathbf{B}_{ij,kl} u_{ij,kl} + z \geq 0 \quad (2.23)$$

where $w_{00}=a_{00}-1$, and $x_{ij}=y_{ij}=I$

Since the absolute value of y and u cannot be larger than 1, the following inequality guarantees that (2.23) is always true, when $y_{ij}=1$:

$$w_{00} + z \geq \sum_{kl \neq 00} |\mathbf{A}_{ij,kl}| + \sum_{kl} |\mathbf{B}_{ij,kl}| \quad (2.24)$$

Similarly, a cell stays the negative saturation region (n), if the following inequality is true:

$$w_{00} + z \leq \sum_{kl \neq 00} |\mathbf{A}_{ij,kl}| + \sum_{kl} |\mathbf{B}_{ij,kl}| \quad (2.25)$$

Note: Rule 6 states that under certain conditions, derived from the template and not the neighborhood pattern only, a cell cannot leave one of the saturation region. Hence, cells, which are in that saturation region, cannot be activated. However, it does not state anything about states, which are in the other saturation region. Those may stay, may leave, according to the local neighborhood pattern.

After defining the necessary condition of the mono-direction cell transition, we examine what is required to guarantee the strictly monotonic cell transition.

Theorem 1. Assuming that only mono-directional cell transition is enabled (Rule 6 satisfied), and $a_{00} > 1$, and the off-center \mathbf{A} template elements are positive, the pixel transition is strictly monotonic.

Proof. We are using an indirect proof here. We will prove the theorem for negative to positive cell transition. Since it is symmetric, it can be proved for the other direction similarly. In t_0 some of the cells leave the negative saturation region, and the spatial-temporal transient starts. Assume that cell (ij) is the first cell in the array, which changes course, and starts heading back to the negative saturation region. Let us denote the time instant with t_1 , when cell (ij) left the negative saturation region, and t_2 , when it changes course.

In t_1 , we know that

$$\dot{x}_{ij}(t_1) = w_{00}y_{ij}(t_1) + \sum_{ij,kl} \mathbf{A}_{ij,kl}y_{ij,kl}(t_1) + \sum_{ij,kl} \mathbf{B}_{ij,kl}u_{ij,kl} + z > 0 \quad (2.26)$$

$$(x_{ij}(t_1) = y_{ij}(t_1) = -1 \text{ because we are in } N)$$

was true, otherwise it would have not left the saturation region. Due to the positive feedback ($w_{00} > 0$), and the assumption that there are no declining output in the array before t_2 , we know that the second derivative of the state was positive in t_1 too. However, in t_2 , the following was already true, because it changed course:

$$\dot{x}_{ij}(t_2) = w_{00}y_{ij}(t_2) + \sum_{ij,kl} \mathbf{A}_{ij,kl}y_{ij,kl}(t_2) + \sum_{ij,kl} \mathbf{B}_{ij,kl}u_{ij,kl} + z < 0. \quad (2.27)$$

$$(x_{ij}(t_2) = y_{ij}(t_2) \text{ because we are in } l)$$

We will show here that it is impossible. Let us see each term in the form:

- $y_{ij}(t_1) < y_{ij}(t_2)$, because the state was rising before t_2 . Hence $w_{00} y_{ij}(t_1) < w_{00} y_{ij}(t_2)$, because w_{00} is positive.

Since the third and fourth terms are not time dependent, only the second term can

$$\sum_{ij,kl}^{kl \neq 00} \mathbf{A}_{ij,kl} y_{ij,kl}(t_2) < \sum_{ij,kl}^{kl \neq 00} \mathbf{A}_{ij,kl} y_{ij,kl}(t_1). \quad (2.28)$$

Since all the template values are positive, this could happen, only if the output value of one or more cells in the neighborhood decreased between t_1 and t_2 . However, this is a contradiction, because we assumed that cell (ij) is the first cell in the array, with declining output. We cannot even say that other cells started to decline at the same time, because we know that the second derivative was positive in t_1 . Hence first, the derivative was supposed to reach the inflexion point (2.26). However, it could have happen only if some neighbors were already declining when the inflexion point was reached. (Note that the state values are continuously derivable functions, because they are representing physical voltages of capacitances.)

Even if we assume that not only one, but more than one cell's output started to decline in t_2 , we will get that there should be one or more cells, with declining output earlier, to initialize the change. Q.E.D

Having analyzed the properties of the wave propagation in CNN, let us switch to the binary activation patterns. Similarly to the uncoupled case, the morphologic properties of the wave-front can be described by binary activation patterns. In this case a binary activation pattern contains two 3x3 patterns which constructed of black pixel positions, white pixel positions, don't care positions, difference position (see later), and a limit number. The first pattern is referring to the static input of the cell, and the second one to the dynamically changing output. A cell is activated in a particular time instant, if the binary activation pattern matches at least as many positions, as the limit number shows. In many cases, the propagation is independent from the input. In such cases the 3x3 pattern referring to the input contains don't cares only. Hence, it is not even included and the binary activation pattern is constructed from a single 3x3 pattern.

Now, we are ready to analyze the process of propagation cell by cell in an example. Consider the binary image in Figure 9. Suppose that we have a propagation type wave, which deletes the ends of single pixel lines. In our example, there is only one active cell at the beginning of the transient, which changes from black to white. When it is changed, its neighbor becomes the end of the line, hence it becomes active, and starts changing to white, and so on... This is the way, how binary waves propagate in a discrete medium.

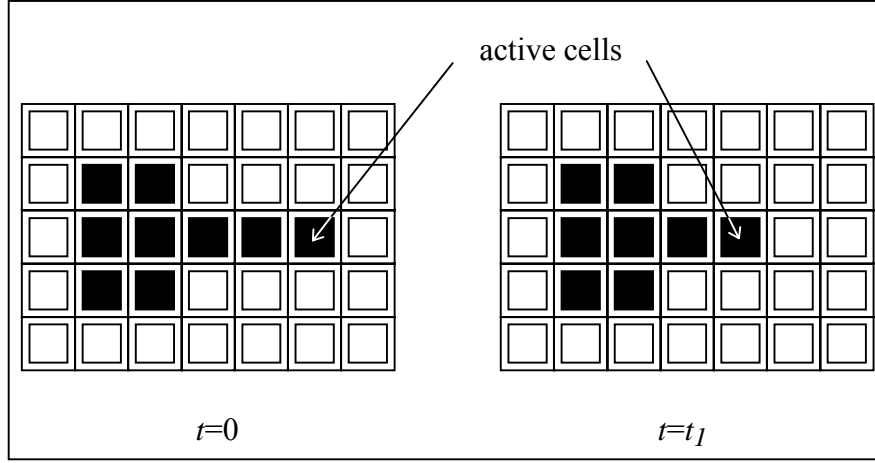


Figure 9. *Propagation example. During the transient there is only one active cell at a time. The single pixel line is deleted pixel-by-pixel.*

Having analyzed the propagation we can discuss the design method. The flow-chart of the design steps of the propagation type templates is shown in Figure 10.

In the first step, we have to describe the global task verbally and with some input-output pairs also. This description must be as precise that based on it, one has to be able to generate the output from an arbitrary input image.

Next, we derive the local rules from the global description of the task. The pixel level rules of the propagation should be derived from the global task, like we did it in example in Figure 9. Then, we can generate the binary activation patterns.

The center element of template A is always a free parameter. Each of the non-don't care elements are commonly given the second free parameter. If the propagation depends on the input too, we have to introduce some further free parameters in template B, according to the activation pattern. In most cases, it is one new free parameter in the center position of template B, but sometimes (according to the priority levels of the activation pattern) there can be more additional free parameter in the neighborhood, as we have seen in the uncoupled case. The bias (z) is the last free parameter.

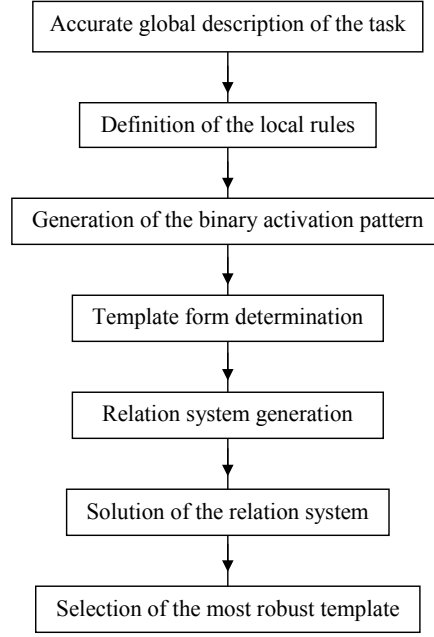


Figure 10. The flowchart of design steps of propagation type templates.

The last three steps are the same as it was in the previous section. For illustrating the design method, here we show two design examples, a simple one, and a more complex one:

Design example C

Task: Generate the left to right horizontal shadow of black objects in binary image.

1. Global description

This is a row-wise problem. If there is a black pixel in a row, all white pixels, which are right to it, should change black, and the rest of the pixels should remain unchanged, as it can be seen in Figure 11.

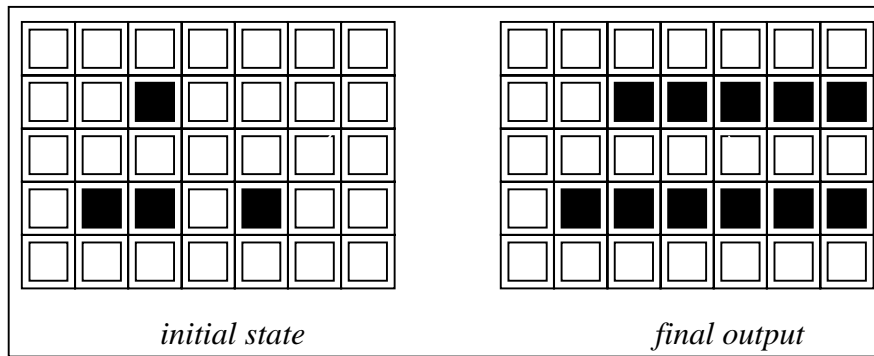


Figure 11. Example for the left to right shadow generation.

2. Local rules

In this task, we have to find the left-most black pixel in each row, and change all white pixels in its left to black. This can be done by starting a black propagation front moving right from each black pixel. Hence, the local rules are:

- (i) a white pixel with a black left neighbor should change to black;
- (ii) the rest of the pixels should not change.

3. Binary activation pattern

The activation pattern belonging to this local rule is the following for the output image:

—	—	—
■	□	—
—	—	—

(The one for input image is fully don't care.) White pixel of such neighborhood condition should change black. (Limit number is 1.)

4. Template form determination

The template form can be derived from the activation pattern and the classifications. The center element of template A (a_{00}) is the first free parameter. There is only one off-center in the activation pattern, which is the second free parameter. Template B is zero, because the propagation is independent of the input. The bias (z) is the third free parameter. The template is sought in the following form:

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ b & a & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad z = i \quad (2.29)$$

5. System of inequalities

In the task description, we allowed white to black transitions only. Therefore, we have to form inequalities on a way that pixels in the positive saturation region should remain inactive in any case (Rule 4), and pixels in the negative saturation region should be activated only when they have a left black neighbor according to the binary activation pattern (Rule 5). These rules can be covered with the following four binary situations:

local pixel arrangement	desired output	state	relation	
	white	inactive	$-a-b+i < -1$	
	black	inactive	$a-b+i > 1$	(2.30)
	black	inactive	$a+b+i > 1$	
	black	active	$-a+b+i > -1$	

The optimized final template is the following:

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad z = 1 \quad (2.31)$$

As we can see, there are no negative off-center \mathbf{A} template elements, and Rule 6 is valid. Hence Theorem 1 will ensure that the cell transitions will be strictly monotonic.

Design example D

Two binary images are given. The first contains some black objects against white background. The second is derived from the first one by changing some black pixels to white. This way some objects become smaller in the second image than in the first one. Those objects, which became smaller in the second image, are considered to be marked. The task is to design a template, which deletes the marked objects and do not affect the rest of the image. If we delete a single pixel of a black object and apply this template, all the black pixels connected to the object will change white, hence the object will be deleted. This template is called Connectivity in the Template Library [23], because it enables to reveal connected components.

1. *Global description*

This is a 2D problem. All black pixels of the marked objects should change white, and the rest of the pixels should remain unchanged. An example can be seen in Figure 12.

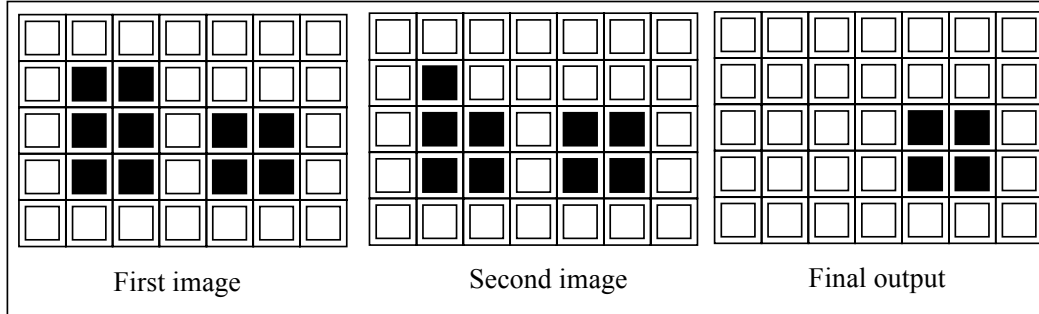


Figure 12. *Example for the Connectivity template. The black object on the left is marked, hence it is supposed to disappear during the transient.*

2. *Local rules*

In this task first, we have to find those pixels, which are black in the first image and white in the second image. From these points we have to start propagation wave-fronts to all directions. The front should propagate on the black pixels only and change them to white. Since the wave front moves on the second image, it should be the initial state and the first image should be the unchanging input. Hence, the local rules are the following:

- (i) change those black pixels to white which have at least one neighboring cell with white output and black input

- (ii) do not change the rest of the pixels.

From this it follows that here the difference of the output and the input matters instead of the output value of the neighboring cells.

3. Binary activation pattern

In this task the activation pattern is constructed from two 3×3 patterns, because the activation depends on both the output and the input. We introduce a new symbol in the activation pattern. The delta symbol (Δ) means that the particular neighbor activates the cell if and only if its output and its input is different. Note that the definition of the task excludes those situations when the output is black and the input is white. A cell becomes active if it has at least one matching neighbor. For simplicity, we used four-cell neighborhood. The activation patterns are shown in Figure 13.

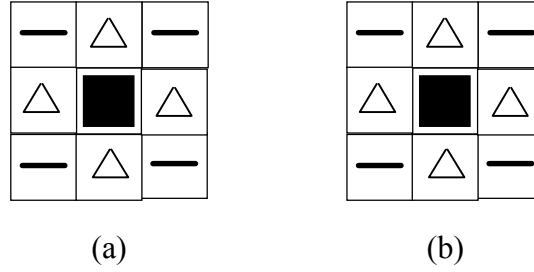


Figure 13. Binary activation pattern for the Connectivity template. (a) shows the output dependency of the activation and (b) shows the input dependency. The delta symbol (Δ) means the different output and input.

4. Template form determination

As usual, the template form can be derived from the activation pattern. The center element of template A (a_{00}) is the first free parameter. The delta operators in the neighborhood effect both template A and template B. A neighbor which has the same input and output (either black or white) does not effect the cell. But if it has black input and white output it activates the cell. Hence, the second free parameter appears in the neighborhood in both templates A and B, but with opposite sign. The center element of template B is the third free parameter, and the bias (z) is the fourth one. The template is sought in the following form:

$$\mathbf{A} = \begin{bmatrix} 0 & b & 0 \\ b & a & b \\ 0 & b & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 & -b & 0 \\ -b & c & -b \\ 0 & -b & 0 \end{bmatrix}, \quad z = i \quad (2.32)$$

5. *System of inequalities*

Since there are only three valid binary input-output combinations here, and 5 matching possibilities, there are 15 different cases. All cases yield an inequality by derived from Rule 4 and 5. The relation set is the following:

output	input	# of matching pixels	state	desired output	relation	
black (+1)	black (+1)	0	inactive	black (+1)	$a+c+i>1$	
black (+1)	black (+1)	1	active	white (-1)	$a-2b+c+i<1$	
black (+1)	black (+1)	2	active	white (-1)	$a-4b+c+i<1$	
black (+1)	black (+1)	3	active	white (-1)	$a-6b+c+i<1$	
black (+1)	black (+1)	4	active	white (-1)	$a-8b+c+i<1$	
white (-1)	black (+1)	0	inactive	white (-1)	$-a+c+i<-1$	
white (-1)	black (+1)	1	inactive	white (-1)	$-a-2b+c+i<-1$	
white (-1)	black (+1)	2	inactive	white (-1)	$-a-4b+c+i<-1$	(2.33)
white (-1)	black (+1)	3	inactive	white (-1)	$-a-6b+c+i<-1$	
white (-1)	black (+1)	4	inactive	white (-1)	$-a-8b+c+i<-1$	
white (-1)	white (-1)	0	inactive	white (-1)	$-a-c+i<-1$	
white (-1)	white (-1)	1	inactive	white (-1)	$-a-2b-c+i<-1$	
white (-1)	white (-1)	2	inactive	white (-1)	$-a-4b-c+i<-1$	
white (-1)	white (-1)	3	inactive	white (-1)	$-a-6b-c+i<-1$	
white (-1)	white (-1)	4	inactive	white (-1)	$-a-8b-c+i<-1$	

The optimized final template is the following:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 3 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 3 & -1 \\ 0 & -1 & 0 \end{bmatrix}, \quad z = -4 \quad (2.34)$$

A complex example can be seen for the propagation of this template. The interesting feature of this example is that it contradicts Minsky's statement. Minsky said that the global connectivity problem cannot be solved by using a locally interconnected network [54].

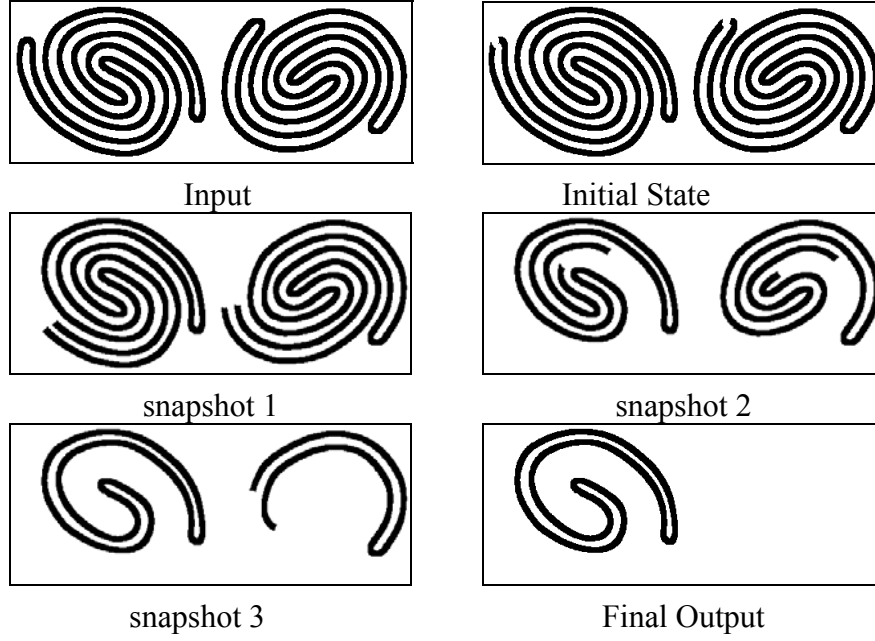


Figure 14. A complex example for the propagation of the connectivity template.

2.3.1 Application range

In this chapter we briefly discuss the three binary-input \rightarrow binary-output propagating CNN template classes (Figure 15), which are important from application point of view, and which can be found in the Template Library [23]. The first two enables pixel transitions in both directions, the third does not. It will be explained here, why the introduced template design algorithm supports mostly the third class, though it provides good results in the first two cases too.

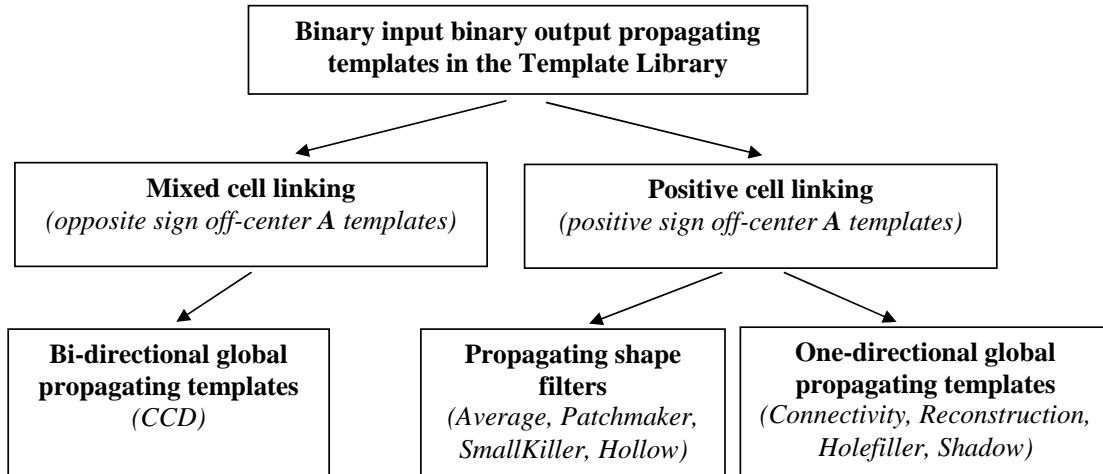


Figure 15. The binary-input \rightarrow binary-output template classes, which can be found in the Template Library.

2.3.1.1 Bi-directional global propagating templates

The best representative of this template group is the Connected Component Detector (CCD) template (2.35). The template has opposite sign off-center **A** template values, which is responsible for its very peculiar dynamic behavior. As it is shown in Figure 16, at the end of the transient, the number of connected horizontal components in each row is represented with the number of black pixels on the right hand.

If we went through the steps of the introduced design method, we would get the template (2.35). However, we cannot prove that the template behaves correctly in the linear region, because its stability is still not proven for those cases, when the array size exceeds 1×3 [53].

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 2 & -1 \\ 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad z = 0 \quad (2.35)$$

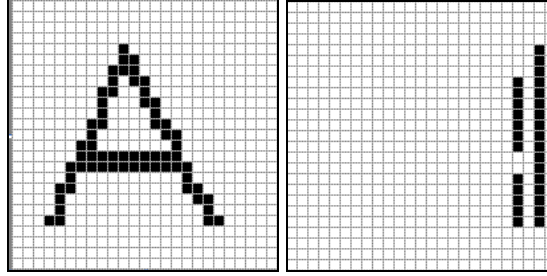


Figure 16. Operation of the CCD template.

2.3.1.2 Propagating shape filters

The most important piece from this template group is the Average template (2.36). It changes a pixel to the dominant color in its neighborhood. For example if a black pixel has three or four white neighbors, it will turn to white, and vice versa. This template makes the boundaries of structured ragged objects smooth. This template class uses positive off-center **A** template values (positive cell-linking) and it is proven to be completely stable [30].

One can use the presented design method for this template class too by defining the global description and the activation pattern. However, exact behavior cannot be derived due to some unpredictable critical race situations which happen in case of processing images with certain patterns. As it is shown in Figure 17 the single “antennas” (black in the top, white in the bottom) are deleted anyway, but on the right side of the black blob in the figure the black pixels has three white neighbors, and the white pixels has three black neighbors. Both of them should change, however only the “fastest” ones can. The final result depends on the local template variation differences and/or thermal noises. Neighboring pixels will “fight” in the linear region, pixel trajectories will not be monotonic, which means that the system of inequalities, derived from the binary situations, cannot predict the final result for each pixel.

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad z = 0 \quad (2.36)$$

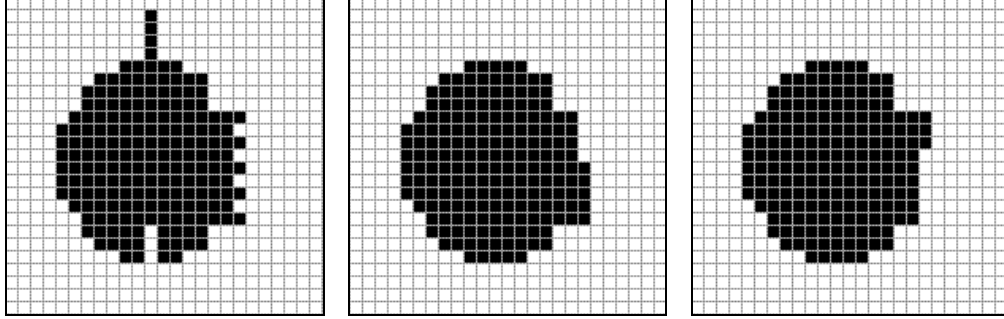


Figure 17. Input (left), and two possible outputs (right) of the average template.

2.3.1.3 Uni-directional global propagating templates

The third template group enables one-directional pixel transients only. Here we use positive off-center template elements too, and the one-directional transient is guaranteed by an appropriate bias term of the template, which satisfies Rule 6. Due to the single direction transients, the critical race situations are completely eliminated, and the transients are strictly monotonic. Thus the final output is fully predictable from the input.

2.4 Conclusions

Direct template design methods have been introduced for propagating and non-propagating type binary input \rightarrow binary output templates. Applying the presented template design method, one can find the robust CNN templates, which can be used either in simulator environment or can be applied to real CNN chips.

The impact of the method can be measured in the scientific literature. The journal paper, described the method [1] was referred over 30 times. The citations were coming from different research teams all over the world. Though the paper was published in 1999, the latest known citation is from 2008, which means that it is still in use.

After the method was developed, we have updated the Template Library [23], and replaced the non-robust templates with robust ones. We have even developed a program – called TEMMASTER or TEMPO [25] – which can synthesize the rules and calculate the optimal template values from the activation patterns. The method is taught in different universities.

3 Virtual processor arrays

The original definition of the CNN [26] assumed a 2D topographic sensor-processor array with mixed-signal processor kernels, which executed various processing steps on 2D data matrices, usually images. The operations, as we have already seen, have been implemented by programmable analog spatial-temporal array dynamics, which were continuous in time and signal, and discrete in space. The implemented mixed-signal CNN chips were champions in capturing and processing images thanks to the ultra-high computational power of the cell array and the embedded optical sensor array. I was the first pioneer in exploring above 10,000 FPS single chip image processing applications [11].

After proving the ultra-high frame-rate operation of the CNN chips, a natural question arose. Is it possible to utilize this extraordinary high computational capability in video processing? As it quickly turned out, the CNN chips applying the original architecture could efficiently process images if their resolution were the same as the processor array size, because the architecture was designed for one pixel per processor topographic data mapping. The straightforward idea, to cut the large image to overlapping tiles, which covers the entire image, and process the tiles one after the other did not work, because the IO time was long, and the large boundaries led to a huge overhead. Though even an analog RAM (ARAM) chip [45] was designed and built to speed up external data communications, its size was still far away from video frames.

Responding to this need, I proposed two architectural solutions for handling early vision problem, and led the design of two others, for foveal post processing purposes. All of these architectures followed the virtual processor array concept [32], what will be introduced in the Section 3.1. The architectures themselves will be described in Sections 3.2, 3.3, 3.4 and 3.5.

3.1 The virtual topographic array concept

If a large number of processors are integrated onto a single chip, the processors should be equipped with local memories, because they cannot reach outside data sources parallel due to obvious pin count limitations. In case of a 2D processor array, the processors are usually arranged on a regular grid, which makes topographical mapping of an image obvious. Figure 18 shows the mapping in a situation, when the data array size is equivalent to the fine-grain processor array size.

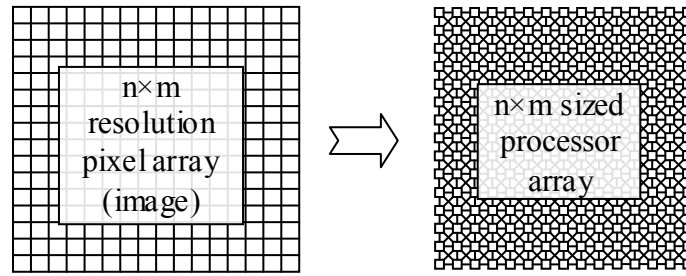


Figure 18. Topographic mapping of an image onto a fine-grain 2D processor array

Though these 2D fine-grain CNN type topographic engines could process images on extraordinary high frame rate (above 10,000 FPS or less than $100\mu\text{s}/\text{frame}$ [11]) the image resolution could not exceed the dimensions of the physical processor array. This means that the physical processor cell size on silicon (from 35×35 up to 75×75 micron [42], [44]) limits the largest feasible array size to about 50k cells. This limit is well below the standard analog video frame sizes and very far from the today standard megapixel digital video formats. Moreover, in many cases, there is no need to process these large format images on ultra-high frame rates, because these images are provided by high resolution imagers serially, which can provide typically images on video rate (30 FPS) anyway. The question is: how to trade resolution for speed, or in other words, *how to convert (silicon) space to (execution) time?*

We have to clearly distinguish two different processing requirement scenarios. In the first one, early image processing problem is addressed. In this case, there is no *a priori* information about the image, hence all parts of the image should be handled the same way, because the location of the relevant parts are not known. In this scenario, usually image enhancement and/or areas of interest identification is done.

In the second scenario, the goal is to perform post deep analysis on certain regions of the image only, because it is assumed that these regions carry all the relevant information of the image. The position and the size of the processed regions (region of interest, ROI) are derived from the results of early image processing. This is an economic solution, because it does not require deep analysis of the entire high-resolution image. On the other hand, assuming an accurate region of interest selection mechanism, we do not lose relevant information. This post-processing approach is called foveal processing, because it mimics the operation of a foveal vision system of primates.

3.1.1 Virtual processor arrays for early vision applications

As we have learned, the root of the problem is that we cannot implement a large enough processor array, which can handle full video frames in one piece. However, we can overcome the problem by introducing a video frame sized virtual processor array, which virtually processes the whole image parallel. Behind the high resolution virtual processor array, an affordable sized physical processor array is performing the calculations. Therefore, only a part of the high-resolution image is topographically mapped and processed at a time.

When a small topographic array processor processes a high resolution image, piece-by-piece, we have to consider two issues. First of all, the data transfer should be optimized. On the other hand, if neighborhood operators are executed, the boundary conditions should be handled properly. This means that we have to be aware of the radius of information required to complete the operations, and use at least as large overlap in the mapping phase as this radius is. However, in some cases, when global propagating type operator is executed (e.g., hole filling) a simple overlapping is not satisfactory, hence multiple scans are required. These issues will be analyzed in Section 4.2.2.1.

The properties of the physical processor array depend on the image sequence type and the required operations. Here we consider early image processing of medium or high resolution analog or digital video images. The images are read out sequentially from an imager line-by-line. The read out pixel train constitutes analog or digital standard video flow formats. By applying an elongated physical processor array, which is exactly as long as an image line is, it can be fed with the pixel train coming out from the imager. In this way, a long and narrow segment of the image, constructed from some consecutive image lines, is mapped at a time onto the processor array (Figure 19). The processed image segment is moving from top to down in discrete steps. The neighboring segments overlap each other to handle boundary conditions. In this way both the IO and the boundary problems are properly handled.

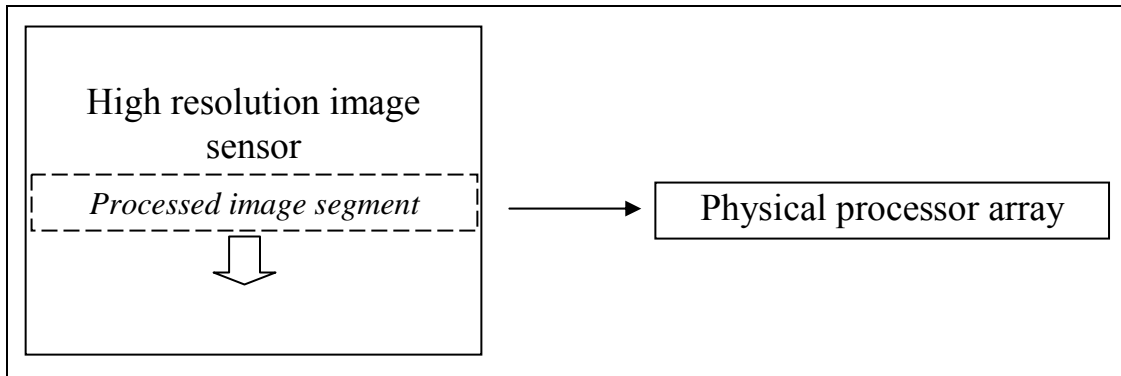


Figure 19. Mapping the high resolution image onto an elongated physical processor for performing early image processing

The first architecture, described in Section 3.2, applies mixed-signal cores. Its specialty is that it can process analog video signals on-the-fly without digitization. The architecture was proposed and patented [4] by myself.

The second architecture, called CASTLE [2][3], is described in Section 3.3. It is based on emulated digital CNN cores. This scalable architecture is designed to be able to process high resolution digital video flows on-the-fly, or can process images, when they are stored in a memory. The architecture was proposed by me [2]. One of its versions was implemented on a full custom digital ASIC [3]. Its derivatives are still used both in the academy [50] and the industry [22].

3.1.2 Virtual processor arrays using foveal approach

It is a well-known phenomenon that the high level information content of an image in most scenes is focused to one or a few areas (regions of interests, ROIs) rather than equally distributed all over the image. Foveal processing is taking advantage of this fact on a way that it focuses attention (spending computational resources) to the relevant areas only. Naturally, it assumes an appropriate ROI identification strategy in the early image processing phase.

Human vision is also based on this phenomenon. Our eyes have roughly a 210° visual field with varying sensor density (Figure 20). The periphery of the retina is a low density monochromatic area. In the periphery, human visual system can identify spatial and temporal irregularities (high contrast pattern, sudden movements) even under low light conditions [55]. The fovea is located in the central area of the visual field. Only this part of the visual field is sensed in colorful high contrast high resolution details. It occupies roughly 3° from the visual field and contains high density color sensors. The output of the fovea is thoroughly analyzed by the farther stages of the human visual pathway.

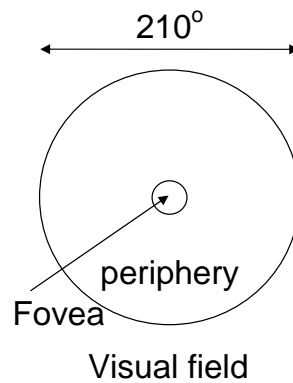


Figure 20. Simplified view of human visual field

We find that we can see a high detailed colorful image in our entire visual field. This is achieved on a way that the fovea – the only area which can perceive this information – is jumping from one point of interest to another. This is called saccadic eye movement [55]. As a contrast, artificial vision systems cannot physically move so quickly due to mechanical limitations, or in many cases, they are not moving at all. To be able to process foveal areas, these vision systems use high resolution, addressable, and zoomable CMOS image sensors, which make possible to read out multiple different sized windows even with different resolution (scale). In this way, a vision system applying the fovea approach can identify and track moving objects by zooming in or zooming out the relevant image parts, according to the scene changes (Figure 21).

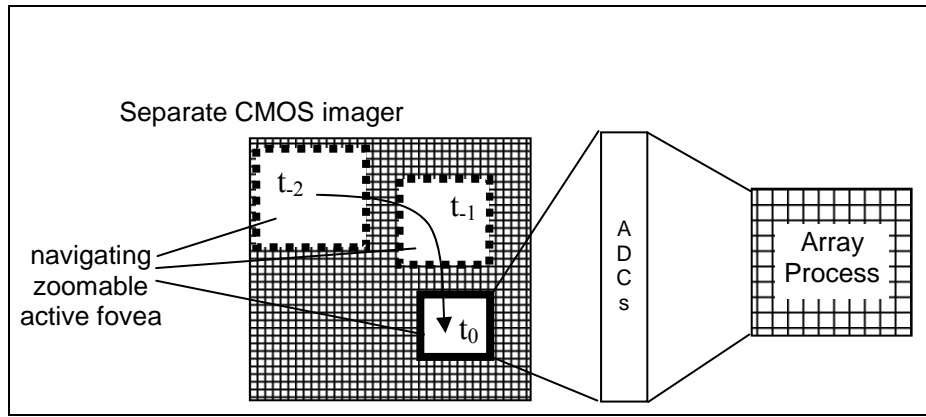


Figure 21. A zoomable fovea is navigating in a high resolution CMOS imager, following the region of interest

Two virtual processor array architectures will be shortly discussed in this dissertation. The first is the Bi-i architecture [10][19], which we introduced jointly with my collage, Dr. Csaba Rekeczky. The Bi-i camera was the first professional camera, which contained a CNN chip as a fovea sensor and co-processor, which enabled it making more than 10,000 visual decisions in a second real-time, which is still unique. Thanks to its high performance, the Bi-i won the product of the year award in the Vision Fair in Stuttgart, Germany in 2003. The Bi-i architecture is described in Section 3.4.

The second architecture, described in Section 3.5, targets a single chip vision system, which combines a fine-grain sensor-processor array as a front-end processor, and a virtual processor array for performing multi-fovea back-end processing. The architecture of the chip is co-designed by two of my collages, Dr Péter Földesy (MTA-SZTAKI), Dr Csaba Rekeczky (Eutecus Inc) and me. Since we need to implement multi-layer, multi-resolution, multi-scale sensor-processor arrays, this ongoing project uses an experimental 3D silicon integration technology.

3.2 Mixed-signal virtual processor array architecture for analog video signal processing

Analog video signal is constructed of a stream of consecutive image lines as they come out from the video sensor device. The mixed-signal virtual processor array was designed to capture and process these signals on-the-fly without digitalization. In general, the architecture can process n incoming video signals, and delivers m outgoing video signals. This makes possible to process RGB image flows or fuse multi-band images coming from different synchronized visual/IR/UV image sensors.

Since the device cannot store entire video frames inside the processor array, it forms long and narrow segments of the images internally by capturing a few consecutive image lines as we have seen in Figure 19. These image segments are then processed by an elongated

processor array. Naturally, these segments should be properly overlapped to avoid artifacts at the boundaries (Figure 22).

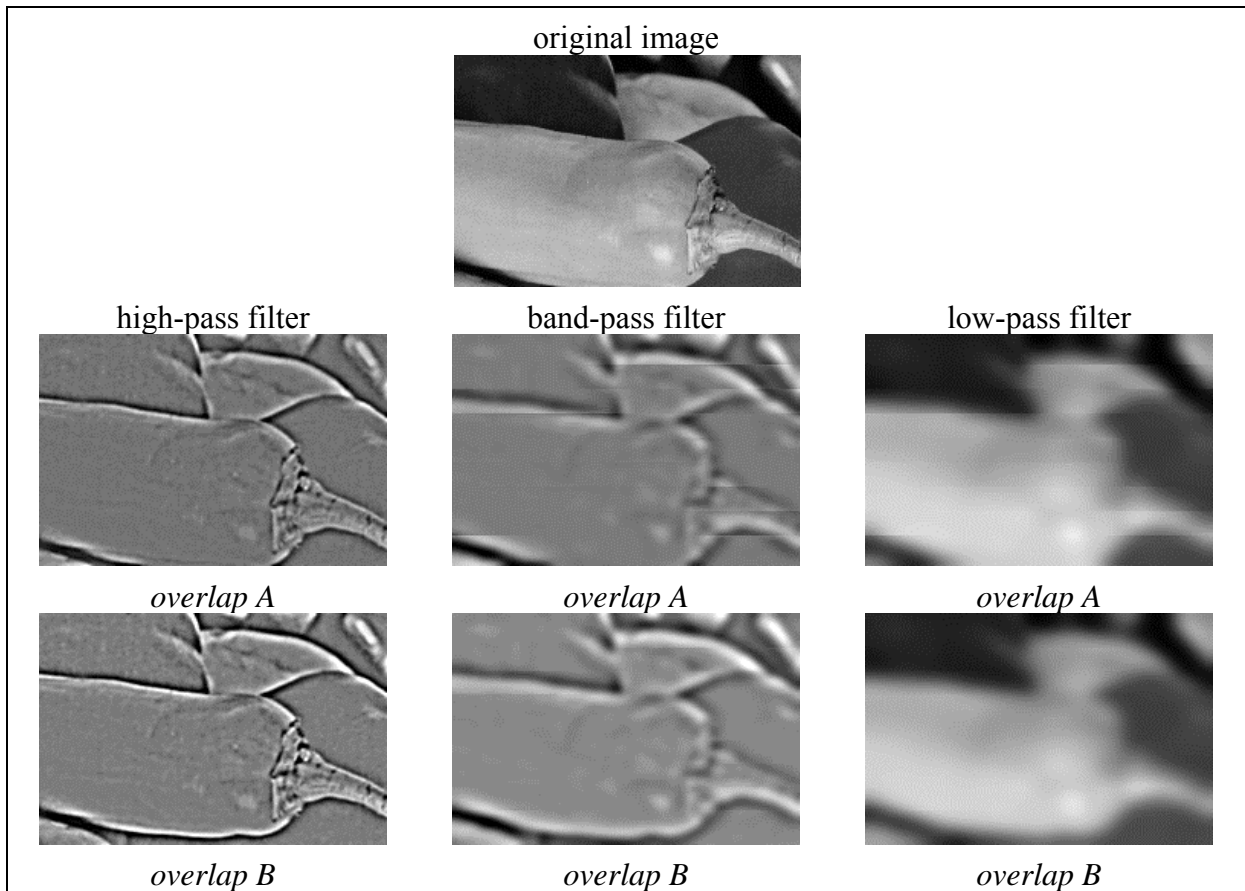


Figure 22. Example for the improper and the proper boundary condition handlings. Multi-scale analysis was calculated on horizontal image stripes with two different overlaps. As it can be seen, the number of the overlapping lines was too low with overlap A for band-pass and low-pass, while it was satisfactory with overlap B for all three cases.

To avoid artifacts caused by the boundary problems, the proposed elongated processor array is divided to three major areas. Two of these areas (the upper and the lower) are dedicated to handle the boundary problem, while the third (middle) area – the main area – calculates the outgoing video lines (Figure 23). The number of the rows in the upper and the lower overlapping areas may be different, because the sphere of influence of the operators might be asymmetric. Moreover, in some cases, the results of the calculations on the preceding segment can be used as upper boundary conditions. Therefore, in these cases it is enough to implement one row of the upper overlapping area to contain the pre-calculated boundary conditions.

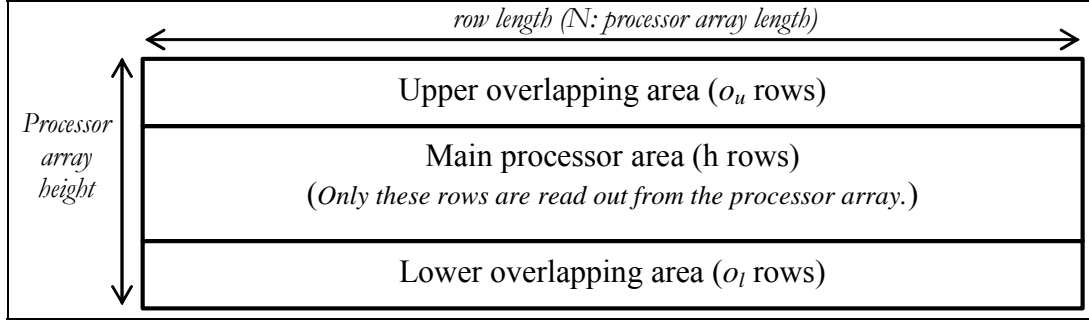


Figure 23. Topology of the elongated video processor

3.2.1 Timing details

First, we introduce the architecture with one incoming, and one outgoing video signals ($n=m=1$). To fulfill real-time concurrent I/O and processing requirements, two extra memory banks, and two line buffers are needed (Figure 24). The incoming and the outgoing video line buffers are two-port analog memories. They can handle both serial and parallel access. On the serial port of the incoming (outgoing) line buffer, it can capture (release) analog video signal. The incoming (outgoing) video line buffer can be read out (filled in) through its parallel port, which is connected to a column-wise parallel bus. This column-wise data bus is responsible for the data communication among the blocks of the system (Figure 24). Its width is N , and it can transfer an entire video line in one cycle.

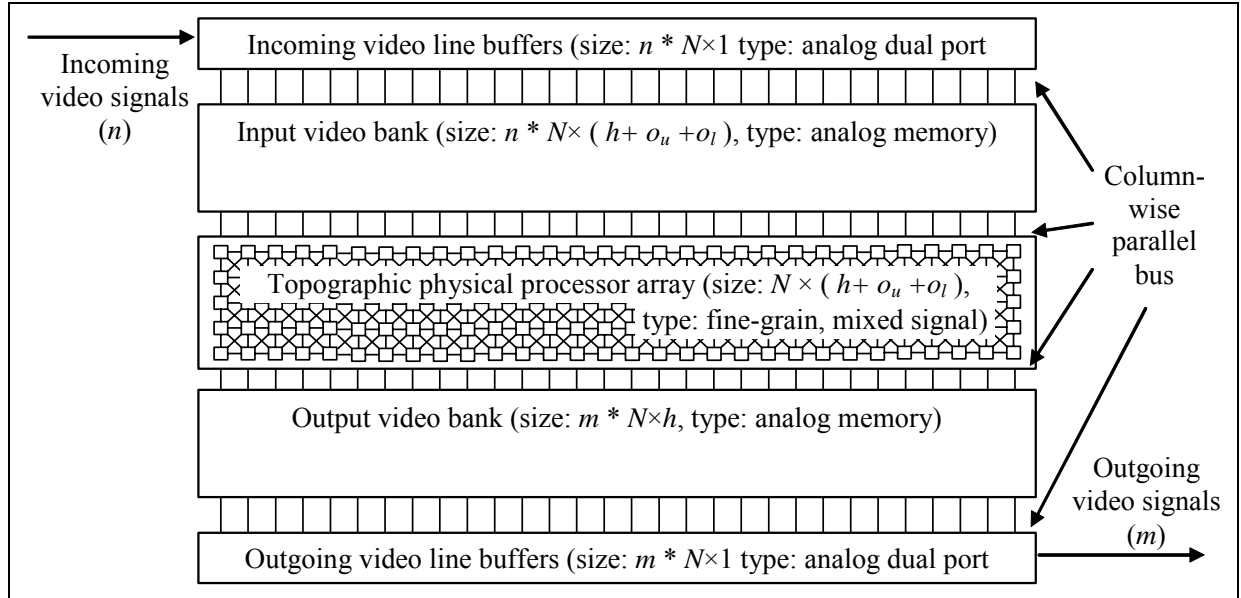


Figure 24. Architecture of the physical processor array

The incoming video buffer having collected an entire video line, sends it to the input memory bank. This is done during the row blanking time, when the video signal does not contain pixel data. The input video bank contains the last $(h + o_u + o_l)$ video lines of the incoming frame. Its full content is transferred to the physical processor array after a new h line segment arrived. This means that the line transfer period and the processing time is equal to:

$$t_p = h * t_l$$

where:

t_p : processing and transfer time;

t_l : line duration ($\sim 64 \mu s$ in PAL or NTSC, and includes $\sim 14 \mu s$ row blanking time);

h : number of rows in the main processor area.

The result of the calculation, the h rows from the main processor area, is transferred to the output video bank at the end of the t_p period, from where the lines are sent to the outgoing line buffer one after the other.

For multiple incoming and outgoing video signal support, the number of the incoming and outgoing line buffers and the input and output video banks are multiplied by n and m , respectively (Figure 24).

3.2.2 Processor options

The mixed-signal processor cells can be either continuous time CNN like devices [44], or discrete time fine-grain mixed-signal [49], [67] ones. These processor arrays can execute an operation in the 10-50 microsecond range. Assuming 15 operations to execute with 9 overlap on an asymmetric way with stored boundary conditions in the upper overlapping area, a 20×640 physical processor array device ($h=10$, $t_p=640 \mu s$) can perform video (speed) processing. This is 12,800 processor plus the memory banks. Since nowadays 16,000 and 25,000 cell mixed-signal fine-grain sensor-processor array devices exists [44], [49], [67] with similar complexity, the introduced architecture is fully feasible.

3.3 Pipe-line virtual digital physical processor array for high resolution image processing

In this section first, we introduce the original CASTLE architecture. Then, in Section 3.3.3 we will analyze the program flow restrictions of the video flow processors.

The CASTLE architecture is an emulated digital implementation of the CNN Universal Machine [24][26][27]. It is a scalable 2D processor array architecture, which processes the images in horizontal stripes. It can handle different bit depths image flows with different speed. The design is introduced for 12, 6 and 1 bit, however, it can be scaled up to 16, 8, 1 bit configuration, according to the sensor readout format and the precision requirements of the applied algorithm.

The main features of the design are as follows:

- The CASTLE architecture is a scaleable 2D digital processor array which processes the images in horizontal stripes.
- It can handle various bit depths image flows with proportionally increasing speed.
- The CASTLE architecture can perform space-variant template operations by applying different template coefficients in different locations according to a predefined map. It is prepared to store a set of 16 templates and makes pixel-by-pixel template selection

possible without execution time overhead.

- The CASTLE architecture digitally emulates Full Range Model CNN [35], using forward Euler integration. It can calculate any 3x3 space variant convolution also.

3.3.1 Principles of operation

In this section, the operation method is described rather than the implementation. First, we introduce the concept of the calculation. This is followed by the description of the equations to be solved, and then the processor core is introduced. Next, cascading is shown, and finally the lower precision operation is described. Two operational modes are introduced. The first is the standard CNN operation mode, which emulates CNN transients with space invariant templates. The second mode supports the usage of different templates in different areas of the image, which makes possible using space variant bias, fixed state, and space variant template operations.

3.3.1.1 Concept of the calculation

The CASTLE processor chip is constructed of an array of $c \times r$ processors (c : column, r : row). The processed image is split to c equal vertical stripes (Figure 25). Each stripe is processed by a column of the processor array. The processor array sweeps through the whole image vertically in each processing path. Each row of the processor array calculates one CNN iteration (convolution) on the image. In this way, in one pass r iterations (convolutions) can be calculated. The first row of the processor array fetches whole image lines from external image source (sensor or memory), and calculates one iteration. Then, it passes the result of the iteration to the second processor row, which does the second iteration, and so on. After the r^{th} iteration is calculated, the last processor row saves the result to external memory, or passes to another on-the-fly device, like another CASTLE chip. Since the calculation of an iteration requires the storage of few lines only, the processor array does not have to store the whole frame, only some lines of it. This relatively small amount of data can be stored in the local memories of the processors.

Note, that the internal processor rows do not need external image data for the operation, which makes possible to scale up the computational performance in the vertical dimension (adding additional processor rows) without any IO increase penalty.

If more iterations are needed, we have to use the processor array more than once. If fewer iterations are enough, we can start a new template operation in the next row.

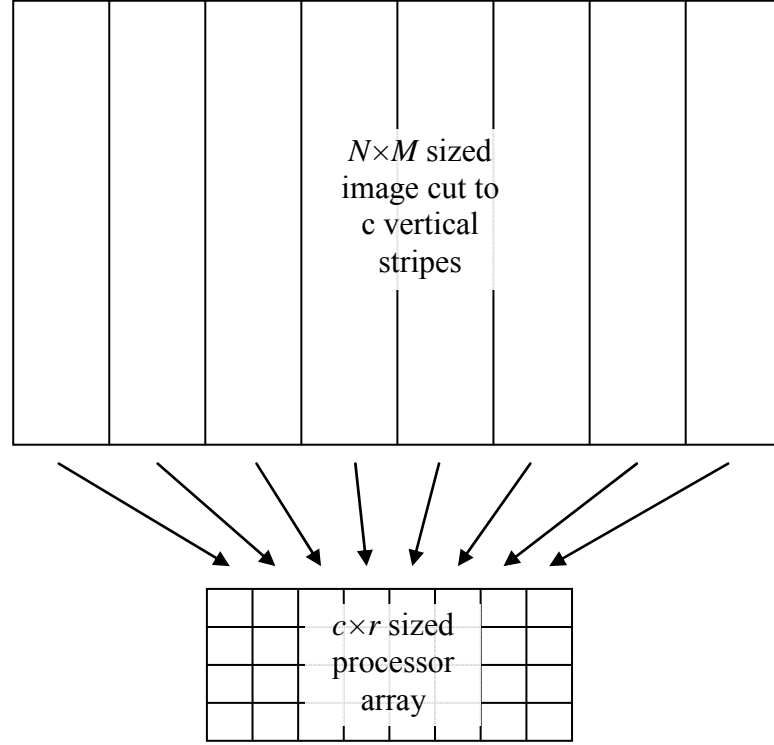


Figure 25. The processed image is split to c vertical stripes. Each stripe is processed by a column of the processor array.

3.3.1.2 Reduction of the CNN equations

In this section, we show the numeric forms, which are used to emulate CNN equation on the CASTLE architecture. As we have seen, the state equation of the original Chua-Yang [24] model is as follows:

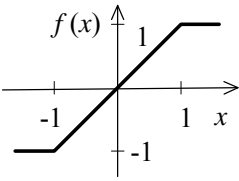
$$\dot{x}_{ij}(t) = \sum_{C(kl) \in N_r(i,j)} \mathbf{A}_{ij,kl} y_{kl}(t) + \sum_{C(kl) \in N_r(i,j)} \mathbf{B}_{ij,kl} u_{kl} + z_{ij} \quad (3.1)$$

Using the forward Euler's formula, we can derive the discretized form of the original state equation (2.1):

$$x_{ij}(k+1) = (1-h)x_{ij}(k) + h \left(\sum_{C(kl) \in N_r(i,j)} \mathbf{A}_{ij,kl} y_{kl}(k) + \sum_{C(kl) \in N_r(i,j)} \mathbf{B}_{ij,kl} u_{kl} + z_{ij} \right) \quad (3.2)$$

To simplify the computation we have to get rid of as many of the variables as possible. First of all we switch from the Chua-Yang model to the Full Signal Range (FSR) [35] model. In this model the state and the output of the CNN are equal. In those cases, when the state value exceeds the saturation level, it is truncated. In this way, the absolute value of the state

variable can not exceed +1. The discretized version of the CNN state equation with FSR model is the following:

$$f(x) = \begin{cases} 1 & \text{if } x > 1 \\ x & \text{if } |x| \leq 1 \\ -1 & \text{if } x < -1 \end{cases}$$


$$x_{ij}(k+1) = f\left((1-h)x_{ij}(k) + h\left(\sum_{C(kl) \in N_r(i,j)} \mathbf{A}_{ij,kl} x_{kl}(k) + \sum_{C(kl) \in N_r(i,j)} \mathbf{B}_{ij,kl} u_{kl} + z_{ij}\right)\right)$$
(3.3)

Thus we combined the x and the y variable by introducing a truncation, which is computationally easy in the digital world. In the next step we include the h and the $(1-h)$ terms into the \mathbf{A} and \mathbf{B} template matrices. It can be done with a simple modification of the original template matrices. The new matrices are as follows:

$$\hat{\mathbf{A}} = \begin{bmatrix} ha_{-1-1} & ha_{-1\ 0} & ha_{-1\ 1} \\ ha_{0-1} & 1-h+ha_{0\ 0} & ha_{0\ 1} \\ ha_{1-1} & ha_{1\ 0} & ha_{1\ 1} \end{bmatrix};$$

$$\hat{\mathbf{B}} = \begin{bmatrix} hb_{-1-1} & hb_{-1\ 0} & hb_{-1\ 1} \\ hb_{0-1} & hb_{0\ 0} & hb_{0\ 1} \\ hb_{1-1} & hb_{1\ 0} & hb_{1\ 1} \end{bmatrix};$$
(3.4)

Using these modified template matrices, the iteration scheme is simplified to a 3x3 convolution, an addition and a truncation:

$$x_{ij}(k+1) = f\left(\sum_{C(kl) \in N_r(i,j)} \hat{\mathbf{A}}_{ij,kl} x_{kl}(k) + g_{ij}\right);$$

$$g_{ij} = \sum_{C(kl) \in N_r(i,j)} \hat{\mathbf{B}}_{ij,kl} u_{kl} + h z_{ij}$$
(3.5)

In the first step, we calculate the constant g_{ij} (lower term in 3.5), then in each iteration we calculate an update (upper term). The only difference is that in the first step we do not have to apply truncation, while in the later iterations we have to apply it.

3.3.2 Architecture description

After introducing the concept of the calculation and the form to be calculated, the architecture is described in the subsequent sections.

3.3.2.1 Image line registers for minimizing I/O

By examining equation (3.5), it turns out that 9 template values, 9 state values, and the constant term are needed for the calculation, and the result must be also saved. It is 20 scalar

data altogether, which obviously cannot be supplied from external sources real-time for each processor and for each convolution.

Most of the scalar data (18 pieces) are needed for the convolution. The template values can be easily stored on-chip, because their number is small (9 pieces). If we store three consecutive image lines (N/c pixel data in each processor's local memory), one new pixel data is needed for each iteration step only. Figure 26 shows the register arrangement, which implements it.

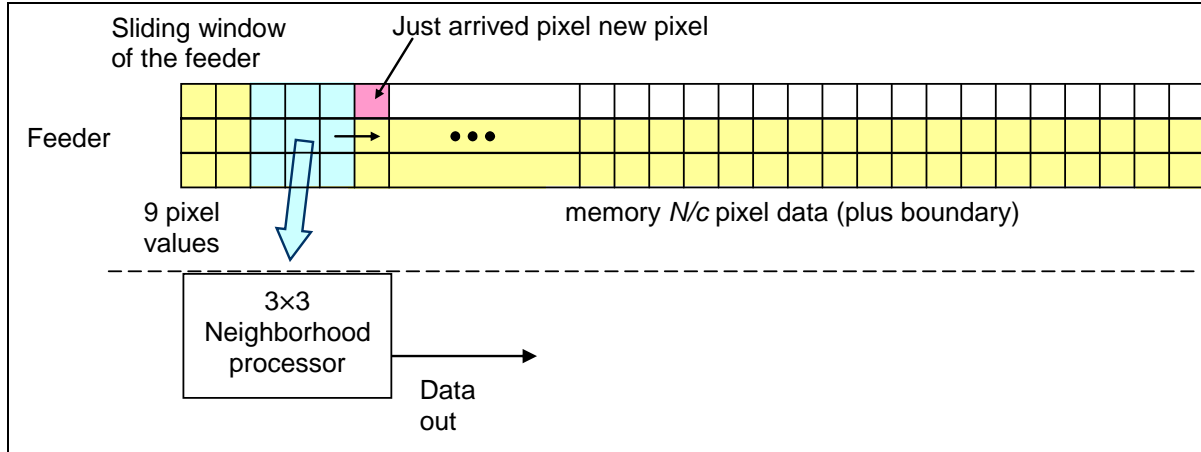


Figure 26. By storing three rows of the image, the number of the I/O can be greatly reduced. The previously stored values are shaded with yellow. The blue square indicates the position, where the convolution is actually calculated. It can be seen that only one new state value is to be fetched during the calculation of an iteration (red).

To further reduce memory requirements, we can simplify the feeder architecture still (Figure 27) producing equivalent input configuration for the neighborhood processor. The simplified feeder contains one non-sliding data latch matrix, and two FIFO lines. The 3x3 non-sliding data latch matrix transfers 9 data to the neighborhood processors in each clock tick, and also shifts the data to the right. A new pixel data is coming from external source, and two others arrives from the end of the FIFOs in each cycle. The two upper pixels from the right columns enter the FIFOs. In this way, we need only one pixel data input, and one pixel data output. The solution of the boundary problem is discussed later.

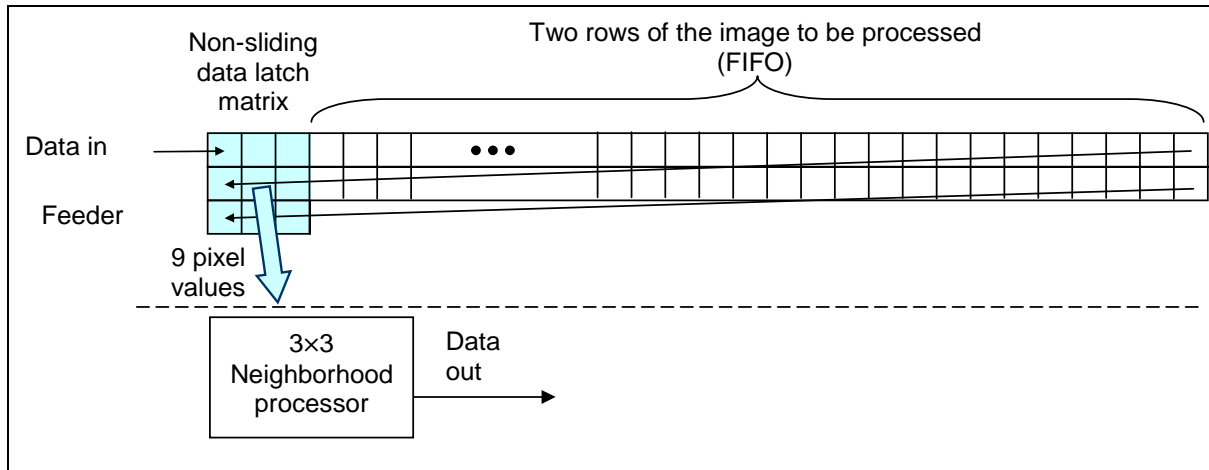


Figure 27. Local memory organization of a processor element

The bus configuration, which delivers the data to the processor, is shown in Figure 28. It contains three input and an output buses. The first input bus loads the state values (x_{ij}), the second brings the constant term (g_{ij}) terms, and the third the template. The output bus passes the result to the FIFO of the next processor row or to an external memory. There is one more bus, called template selector bus (TS bus), indicated. This is used when the template is space-variant, or when fixed state map is applied. The usage of this bus is described later.

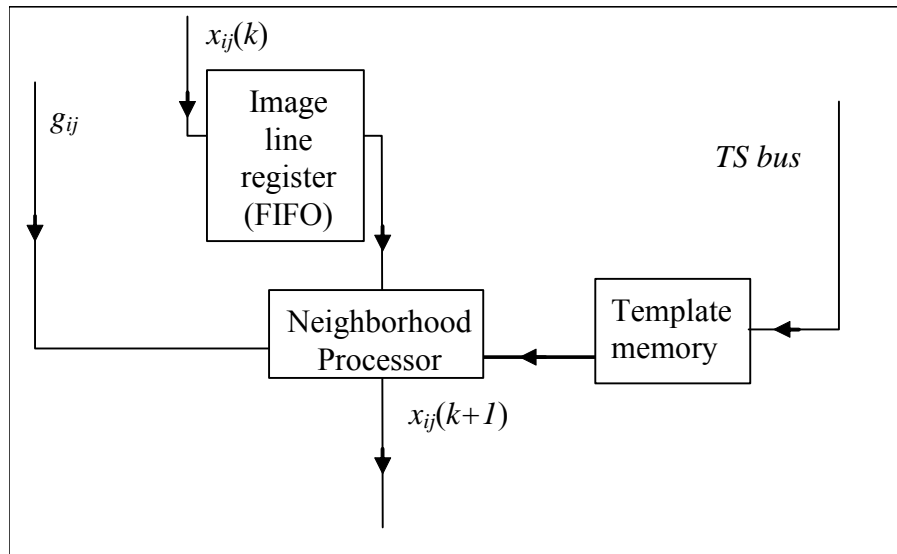


Figure 28 The data bus arrangement of a processor unit

3.3.2.2 Description of a single processing core

The processor core has to calculate a 3×3 convolution, an addition and a truncation, that is 9 multiplications and 9 additions and a truncation altogether. The proposed processor core contains 3 multipliers, and 3 adders. Their arrangement is shown in Figure 29. The calculation of an update is done in 3 phases. In the first phase ADDER #1 receives data from a multiplier and adds it up with g_{ij} , or with the constant value hz_{ij} according to equation (3.5). In the following two phases it receives the output of ADDER #3 through the feedback loop. In each

phases the multipliers calculate a 3×1 convolution. The result appears on the output of ADDER #3 by the end of the third phase.

The processor core requires 3 pixel values and 3 template values at a time. These values are provided via internal parallel busses (Figure 29).

3.3.2.3 Template selector map

CNN operations can be either uniform or non-uniform in space. Uniform operations apply space invariant templates, which means that the same operation is executed in every location of the image. On the contrary, spatially non-uniform operations may apply different templates in different locations on the image. This can be used for example to stop propagations, or to perform different kinds of operations on image parts with different contents. The different areas of the image can be marked with binary masks.

For supporting spatially non-uniform computation, CASTLE can store 16 arbitrary 3×3 template matrices in each processing unit. Each ij position of an image can be convoluted with any template matrix of these 16. The template selection is done by using a template selector map. This map has the same size, as the image. Each binary value of this map (m_{ij}) addresses a template stored in the template memory. The template selector map arrives through the template selector bus synchronized with the state bus.

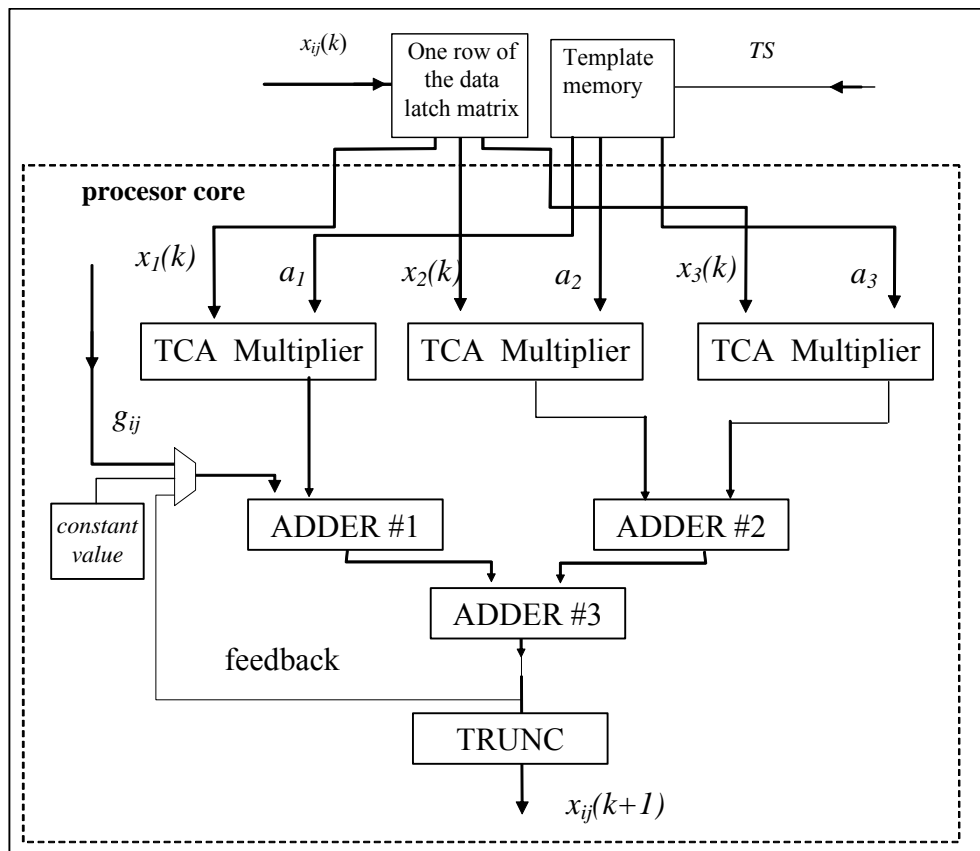


Figure 29 The block diagram of the processor core.

Using the template selector bus, we can implement the fixed-state concept also, which is actually a subset of the space variant template. Fixed state means that the state/output of the CNN is frozen in certain locations of the image selectively. In those positions, where we want to avoid the modification of the image we apply the following template matrix:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

In those cases, when the whole image is processed with the same template matrix and no fixed-state is applied, the template selector bus is not used, and the particular template is selected via global lines.

3.3.2.4 Cascading the processors

The processors are cascaded in both vertically and horizontally, to avoid boundary problems. As it was shown in Figure 25, each processor column is dealing with a separate vertical image stripe. On the other hand, each processor row calculates a new update on the image. First, we describe the horizontal cascading, then the vertical one. We describe them separately, because their roles are totally different.

Horizontal cascading

When calculating convolution on an image, we have to know the surrounding pixel values at each pixel position. It is straightforward, if the image is handled as a single large array, but in our case, the image is split and the image stripes are processed separately by different processor units. To avoid boundary problems at the internal edges of the image, the values at the internal boundaries (at the borders of the vertical stripes) should be exchanged.

This exchange can be achieved by introducing two new columns in the image line registers, one at the left ends, and one at the right ones. The neighboring processor units exchange pixel values as it is shown in Figure 30. The exchanged data fills the newly introduced register columns. This exchange is completed in the row blanking periods, when no data is arriving from the sensor. These periods occur between every two lines in the digital image flow.

Boundary problems cannot be avoided at the external boundaries, because the surrounding pixels are not known there. If we want to avoid the reduction of the image size, we have to introduce an outer frame around the frames. There are two strategies to fill this frame. One possible way is to duplicate all pixel values at the boundary, the other one is to fill the frame with a constant value. Certainly, this frame appears on the horizontal boundaries of the frames also, but that does not require any extra hardware, only the external generation and feeding of the boundary lines in the time period, which occurs between the frames.

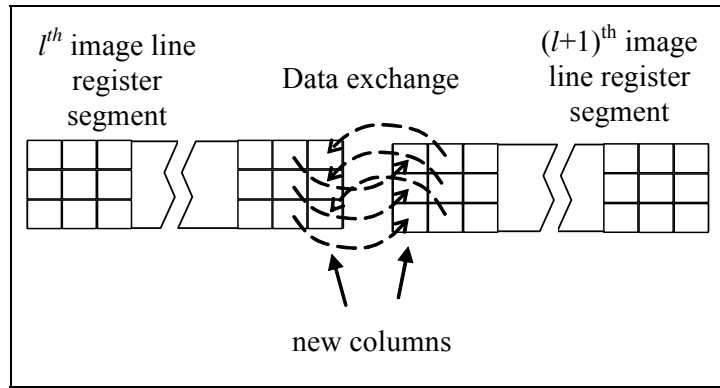


Figure 30 Cascading the processor units horizontally. The length of the image line registers is increased, and data is transferred to the new columns from the neighboring register.

Vertical cascading

The vertical cascading is simpler than the horizontal one, because there are no internal boundary issues. The input (g_{ij}), the state ($x_{ij}(k)$), and the template bus arrives from external pin to processors in the first processor row. In standard CNN operation mode, the state is updated and the data, carried by the two other buses is unchanged. In the middle layers, image data is passed from one processor rows to the others. In the last row, the processed image data is sent out from the device.

3.3.2.5 Variable bit depth arithmetic processor core

Different operators require different computational accuracy. Image processing, especially early image processing does not require high precision, because the incoming data is between 6 to 12 bits. We have made an analysis on the operators listed in the CNN template library [23] and found that 82% of those operators which handle grayscale images can be accurately calculated on 12 bits, and many of them gives correct results even on 6 bits.

Therefore, we proposed to implement reconfigurable processor cores with variable 12 bit and 6 bit data representation in the CASTLE architecture. The image line registers and the arithmetic cores were designed on a way that they can be used in both resolutions.

We have already shown the processor unit structure in 12 bit precision mode. The 6 bit mode uses the same I/O buses, but in this case two pixel value is transferred at a time. The internal data register bank is physically the same, but here two pixels are stored in a 12 bit register. In 6 bit mode, the internal word lengths of the adder and the multiplier are half, than in 12 bit mode. This gives the possibility to use reconfigurable units. In the 6 bit mode, each multiplier and adder are split into two independent units. Thus, a processor core can calculate two convolutions in 6 bit mode. Figure 31 shows the processor core schematics both in 12 and 6 bit modes.

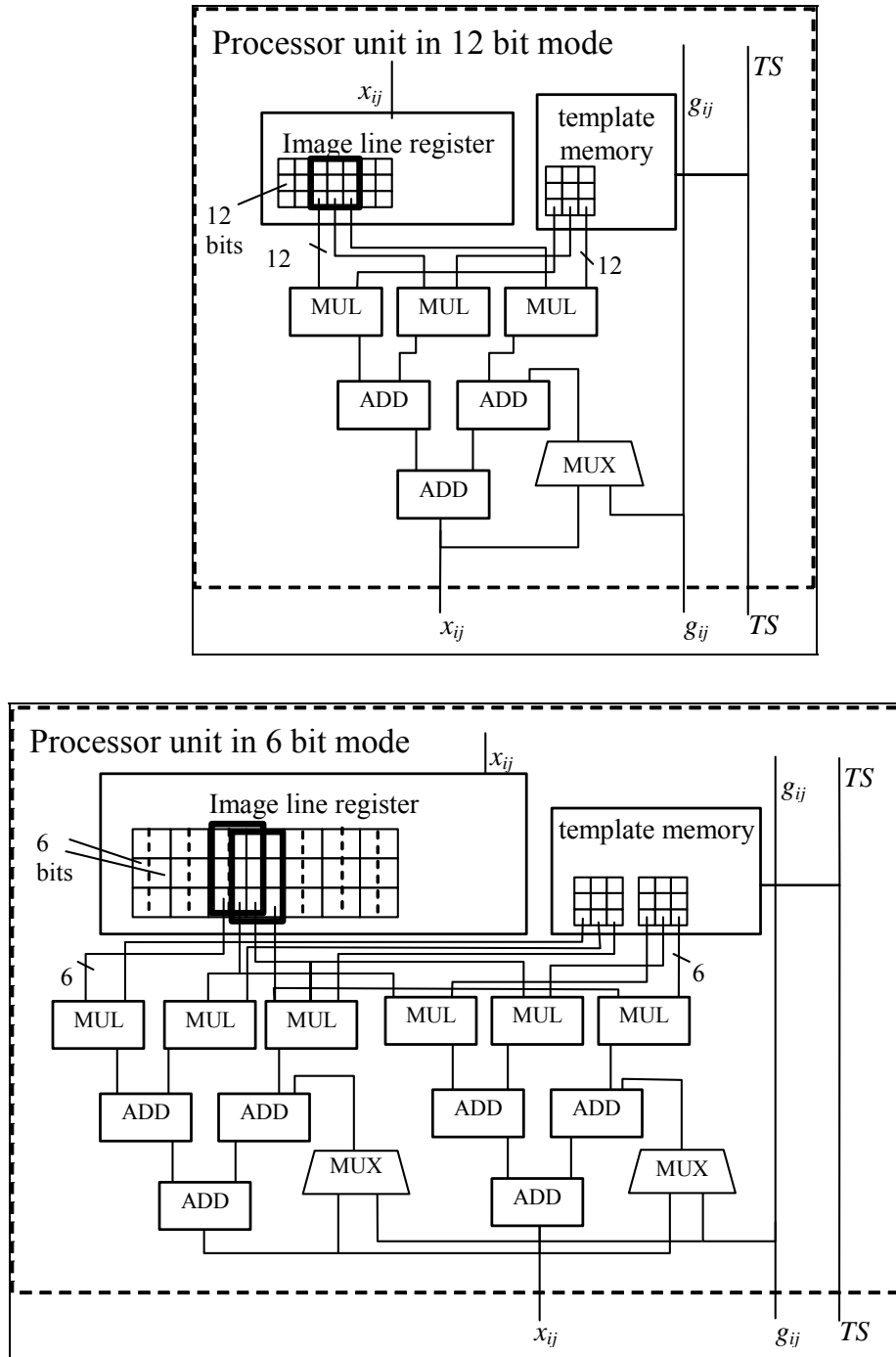


Figure 31 Reconfigured processor core. The image line registers, the multipliers and the adders can be split.

The g_{ij} and the x_{ij} buses carry two 6 bit pixel values instead of one 12 bit at a time. The template selector bus carries two times 4 bit at a time, each selects a template in the template memory. The template value is stored in 12 bits in 12 bit mode, and in 6 bits in 6 bit mode. The horizontal cascading works with the same data exchanging method what we have seen in the 12 bit mode.

3.3.2.6 Binary morphologic processor core

The other large set of CNN operators are the binary input-binary output ones. They can be very efficiently calculated with binary processors, while their calculation efficiency on arithmetic processors is poor. Since the binary operators are heavily used in most image processing applications, it seemed (to be) worth to include binary morphologic processor cores to the CASTLE.

For implementing 1 bit mode in the CASTLE architecture, we have to introduce a logic processor sub-unit in each processor unit. In this way, $k \times l$ logic processor sub-units work parallel in this mode, like in the 12 bit mode. The units use the same internal and external busses, what was introduced in the 12 bit mode. Using the proposed logic sub-unit, we can implement most of the binary input-binary output template functions.

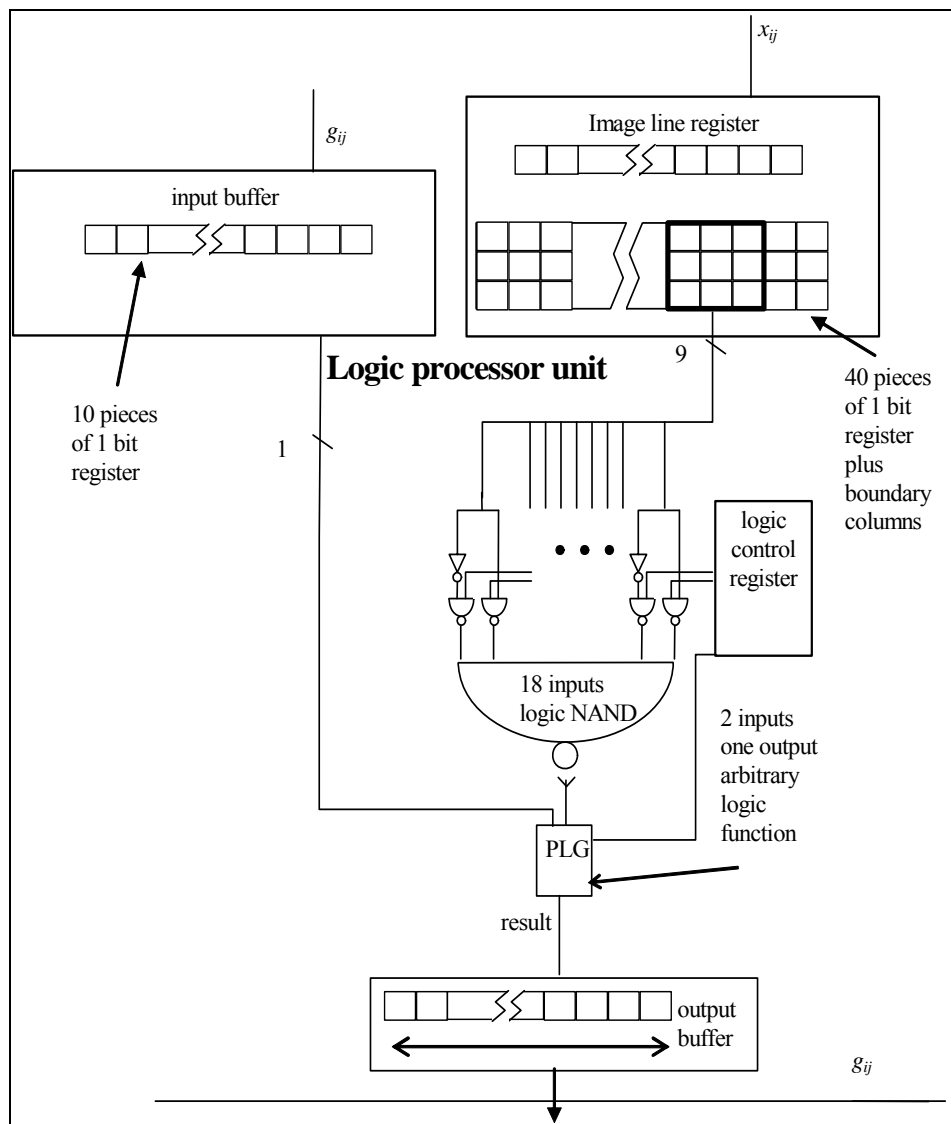


Figure 32 The logic processor sub-unit

The block diagram of the logic processor sub-unit can be seen in Figure 32. The sub-unit receives two binary images, processes them, and transfers the result. It implements a 10 inputs one output logic function. 9 inputs comes from a 3×3 location of the first image (transferred by x_{ij} bus), and the 10th one comes from the second image. The two input images are pixel synchronized. The two images are the state and the input in propagating type template operations with non-zero B template (e.g. connectivity, hole filler, reconstruction, etc).

Three rows of the first image are stored in an image line register bank (like in the arithmetic unit), because we need a 3×3 location of the pixels. The image line registers are cascaded with their horizontal neighbors, similarly to the 12 bit case. The result is collected in the output register.

The logic processor core contains an 18 inputs logic NAND gate. Each logic pixel value from the 3×3 location can be connected to this NAND gate in normal or in inverted form, governed by the Logic Control Register. The result of the NAND gate can be modified with the Programmable Logic Gate (PLG) module. This module applies an arbitrary two input-one output function on to the 10th input value (coming from the second image) and the output of the NAND gate. Since this processor unit has relatively low complexity (compared to the arithmetic unit) it can run much higher clock rate.

The logic processor sub-unit is controlled by the logic control register. This register tells whether a pixel data is used in normal or in inverted form from the 3×3 location, or it is not used at all. The control register also contains the program of the programmable logic gate (PLG). Here we show an example for the implementation of a basic template.

Example: Hole filler template

$$\text{Template: } \mathbf{A} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 3 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad z = -1$$

Function: Changes a black pixel to white in the first image (state) if it has at least one white neighbor, AND the second input image is white in the current position.

Implementation: inputs of the NAND gate: $\begin{bmatrix} - & n & - \\ n & n & n \\ - & n & - \end{bmatrix}$ (n : normal, i : inverted; $-$: not used)

If there are at least one white neighbor, the output of the NAND gate is 1 otherwise 0. If it is 1, the final output should be the same, as the pixel value from the second input image. If

it is 0, the final output should be 1. The logic truth table of the PLG module and an example is shown in Figure 33.

		Output of the NAND gate	
		0	1
pixel value from second image	0	1 (black)	0 (white)
	1	1 (black)	1 (black)

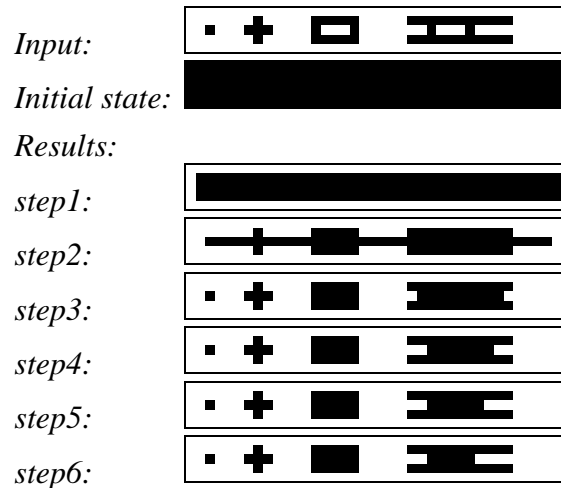


Figure 33 Example of the operation of the 1 bit processors of the CASTLE architecture. The truth table of the hole finder operator is shown above, and the consecutive steps of the image update sequence is shown below.

3.3.3 Program flow considerations

Unlike the full topographic processor arrays, both of the previously introduced pass-through type (or row-wise) virtual processor arrays process only one part of the image at a time (a horizontal stripe). This implies two natural limitations in their applications. The first is that the implementation of a global (propagation or wave type) instruction requires extra care, because in some cases, the final result of the computation depends on distant parts of the image, which is not in the processed horizontal stripe. However, in case of certain propagating operator classes, the row-wise execution with these pipe line architectures leads to significant efficiency increase compared to the full topographic processor arrays. These efficiency questions will be analyzed in Section 4.2.

Other problem arises with the intra-frame (image) content-dependent conditional branches. Intra-frame content-dependent conditional branch means that a calculated parameter of the processed image determines branches or parameters in the further program flow of the calculations (Figure 34a). Generally, we need to process the entire frame to calculate the requested parameter or branching condition. Therefore, the calculation of the next operation

cannot be started before the last the previous operation is completed on the entire frame, because the conditions of the branching or the process arguments are unknown before. Since these processors cannot store more than a few lines of the image in internal memories, an external frame-buffer should be included in the design at each conditional branches (Figure 34b), which significantly increases memory requirements, latency, and circuit complexity.

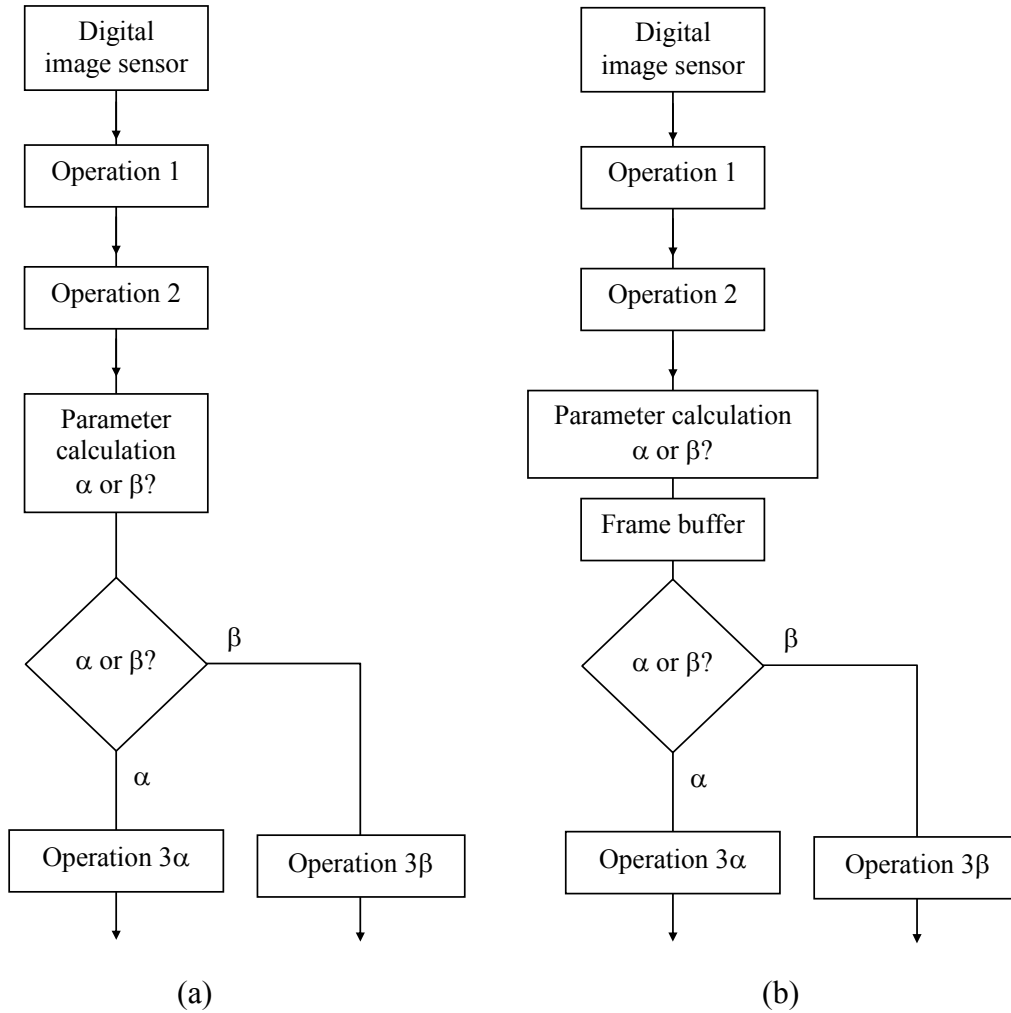


Figure 34. Original program flow (a) and the implementable program flow with an additional frame buffer memory (b)

3.4 Bi-i, a foveal processor architecture based camera system

The Bi-i camera (2002) [8][9][19] (Figure 35) was the first professional camera, with integrated CNN technology. It applied the ACE16k chip [42], which is a 128x128 sized CNN chip, designed by Gustavo Linnan from Angel Rodríguez-Vázquez' team (IMSE CNM, Seville, Spain). The Bi-i is an embedded camera computer, which was designed to be able to make high-speed visual decisions in standalone mode, within the camera head. This means that it can capture images, make decisions, and send over a decision report only, rather than entire image flows. To support these requirements the Bi-i camera applies 4 major components (Figure 36).

- **ACE16k CNN chip.** It is a topographic sensor-processor array, which can capture and process images in ultra-high speed. Its special feature is that each pixel element is corresponded with a mixed-signal processing elements. These high number (above 16000) fully programmable processing elements deliver an extraordinary computational capability. Though it can reach its top performance (above 10,000 FPS), when it both captures and processes the images, because its full-frame grayscale IO bandwidth is limited to 4000 FPS, it can be used as a co-processor too. In co-processor mode, the image is received electrically, rather than optically.
- **Megapixel CMOS sensor.** In a foveal camera system, the high resolution sensor should be able to support multi-scale region of interest (ROI) readout. This means that arbitrary sized and positioned window of the image can be read out, rather than the whole frame. Moreover, the image can be subsampled on the sensor readout level. This is very important, because the image sensors are read out with a fixed pixel rate, hence the frame-rate is proportional with the number of read out pixels. Typical readout time of a full frame is below video speed. The only way to reach higher frame-rates is to drop the pixel count.
- **High-end DSP with memory.** The highest performance Texas DSP was applied in the Bi-i. It has three independent communication channels, to be able to communicate with the three other major components of the Bi-i. Typical operation mode of the system is that the high performance ACE16k chip calculates the computationally demanding parts of the image processing. During this phase, the 2D image is converted to 1D feature vectors. The DSP is used to evaluate this reduced dimension data, and to make final decisions.
- **Communication processor.** A standalone embedded system needs to communicate with other remote computers. However, such a communication requires a complete operating system (OS). If an OS is implemented on the DSP, it loses its ability to be real-time, which is essential to serve in ultra-high speed applications, moreover, it loses significant computational performance. Therefore, a communication processor was applied for handling the external communication. Such a communication processor contains a low performance entire computer including processor, memory, flash, ethernet and other communication periphery.



Figure 35. The Bi-i camera. It has two optical inputs, a low resolution, ultra-high speed focal-plane sensor-processor device (ACE16k chip) and a high resolution CMOS sensor with ROI.

The Bi-i supports three operation modes: a high-resolution (megapixel) low speed (below video speed) mode, a low resolution (128x128) ultra-high speed (above 10,000 FPS) mode, and a virtual high resolution (megapixel) high speed (100-1000 FPS) mode, which combines the previous two, applying the virtual foveal processor array concept. These three modes are briefly summarized in the following subsections.

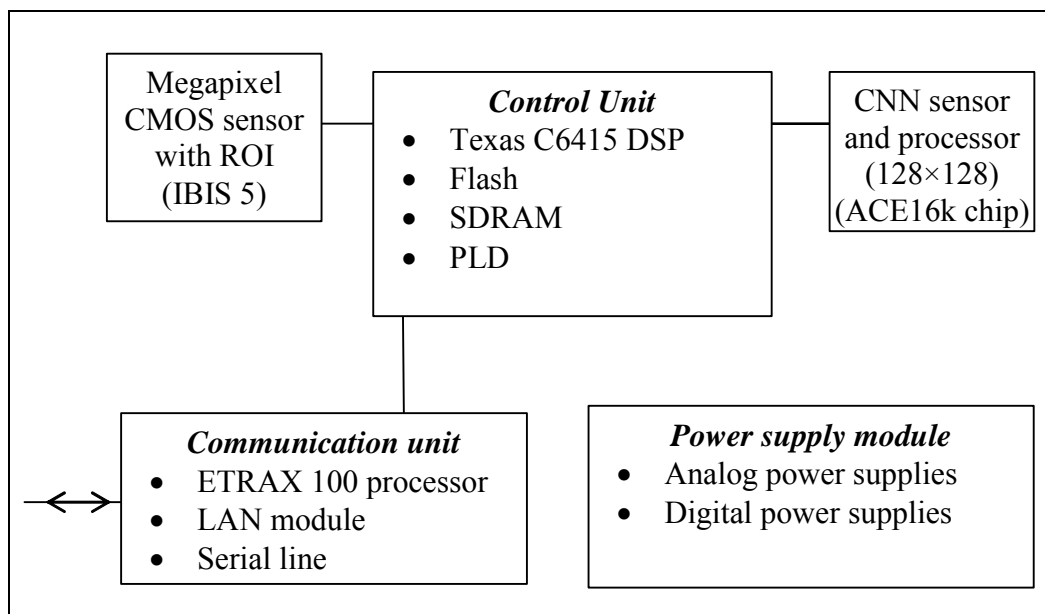


Figure 36. Block diagram of the Bi-i camera

3.4.1 Low resolution (128×128) ultra high speed mode of the Bi-i

The first operation mode of the Bi-i is a low resolution (128×128), ultra high-speed (above 10,000 FPS, or less than 100μs) mode. In this case, it uses the ACE16k chip both for capturing and processing the images. This CNN chip can perform an image processing operation (template) in 2-10 μs, while its external grayscale image IO takes 250μs. Besides the relatively slow grayscale IO, the ACE16k chip has two other output channels. The first is a single bit IO, which tells whether the a binary image is pure black, or it contains at least one white pixel. The other is a vector output, which delivers the coordinates of the black pixels against white background. The first can be used for present or absent kind of decisions, while the second for gaining position information too.

To be able to reach ultra-high speed, all the processing steps, including decisions, are supposed to be executed inside the ACE16k chip, and some present/absent or position readout can be used. Some typical problems which can be efficiently solved with this technology are as follows:

- Event detection;
- Present or absence in non-trivial cases;
- Size, shape and orientation classification;
- Position detection;
- Single or multiple object tracking.

Figure 37 shows an example, where the Bi-i is classifying small objects based on their size and shape above 10,000 FPS.



Figure 37. Bi-i camera, classifying objects above 10,000 FPS.

3.4.2 High resolution (megapixel) video speed mode of the Bi-i

In the second operation mode the Bi-i camera uses its high resolution CMOS sensor as an input device. The captured images are processed jointly by the ACE16k and the DSP. To be able to use the ACE16k chip, the megapixel image should be cut to 64 (8x8) or 81 (9x9) overlapping segments, and the segments should be processed one after the other. Due to the relatively long IO time, processing of a segment takes 600-700 μ s. Hence, the overall processing speed of a frame will be in the 15-25 FPS range, which is balanced with the input rate of the megapixel sensor.

3.4.3 Virtual high resolution (megapixel) high speed mode of the Bi-i

The third mode of the Bi-i camera is the foveal mode. In this mode, the Bi-i uses its megapixel optical sensor, but it does not read out the full frame, rather some regions of interest (ROIs). These regions can be in arbitrary positions. Their size can be arbitrary too, however, we have to keep it in mind that the ACE16k chip will have to process them. Hence, we should be able to put them together to form 128x128 sized images.

The windows can be scaled. This is practically a subsampling of the image already on the sensor level. In case of scale 1:2, technically this means that every odd pixels are read out from every odd rows. The even pixels from the odd rows and the entire even rows are discarded, hence, the readout time is reduced to its $1/4^{\text{th}}$. Certainly, the image is less detailed, but these scaled images are perfectly suitable to identify the locations of the regions of interest in most cases.

Typical application here is, if we need to follow a few different moving objects in the scene. In this case, one or a few downsampled image initially identifies the location of the objects. Then, we zoom into the picture, exactly to those locations, where the objects are, and read out 1:1 scale windows with the objects in central position. The windows are processed one after the other. During this process, besides the exact location of the objects, the characteristic features (grayness, size, various shape descriptors, orientation, etc) are extracted too. Then, the DSP builds up a database from these feature vectors, which makes possible to identify these objects, calculate their kinetic parameters, and make predictions of their next locations. If these predictions are accurate enough, there is no need to make multi-scale search for the objects in each period.

This multi-scale, multi-fovea virtual processing approach makes possible to maintain both high frame-rate and high resolution without losing relevant information. Some examples, where we reached 100-1000 FPS by applying this method are described in [9] and [19].

3.5 VISCUBE, a foveal processor architecture based vision chip

While the Bi-i is a general purpose, high-speed industrial camera computer, the VISCUBE is a low-power visual navigation chip [15] designed to operate on moving platforms. According to its target specifications, it has to be able to capture 1000-5000 medium resolution images in a second and perform image registrations real-time. To be able to fulfill these requirements, it has to apply two such technologies, which are beyond the state-of-the-art, and which are scientifically relevant (2008). The first is, that it combines a fine-grain, mixed-signal, medium resolution sensor-processor array operating in the focal-plane, and a virtual digital processor array operating in foveal mode, on a single chip. To be able to implement these two processor arrays in a single silicon chip, novel 3D silicon integration technology is required, which is still in experimental phase.

3.5.1 VISCUBE architecture

VISCUBE is a focal-plane sensor-processor chip [15]. It contains one sensor and two processor arrays, which are implemented on four physical (silicon) layers. The top silicon layer contains a sensor array. The second silicon layer contains a fine-grain mixed-signal processor array, which is designed by Professor Angel Rodríguez Vázquez' team in Seville, Spain. The third and the fourth silicon layers contain a digital foveal processor array and its memory. Figure 38 shows the high-level block diagram of the architecture and its foreseen implementation in a multi-layer silicon chip. The VISCUBE is a scalable architecture. An embodiment of the VISCUBE architecture will be fabricated in fall 2009, with 320x240 sensor resolution.

The fabrication of the VISCUBE is done in two steps. First the three lower silicon layers (tier A, B, C) are fabricated and integrated using through silicon via (TSV) technology [69]. Then, the sensor layer on the top is fabricated separately, and connected to the lower layers using bump bonding technology [70].

3.5.1.1 Sensor layer

The sensor of the VISCUBE is implemented on its top silicon layer. It is a back illuminated sensor [70]. The sensor layer contains the photodiodes only, hence close to 100% fill factor can be reached achieved. All the other circuitry (amplifiers, switches, reset circuit etc.) will be implemented on the mixed-signal layer. This means that all the photodiodes require a connection to Tier C parallel, because the photocurrents are integrated on that layer.

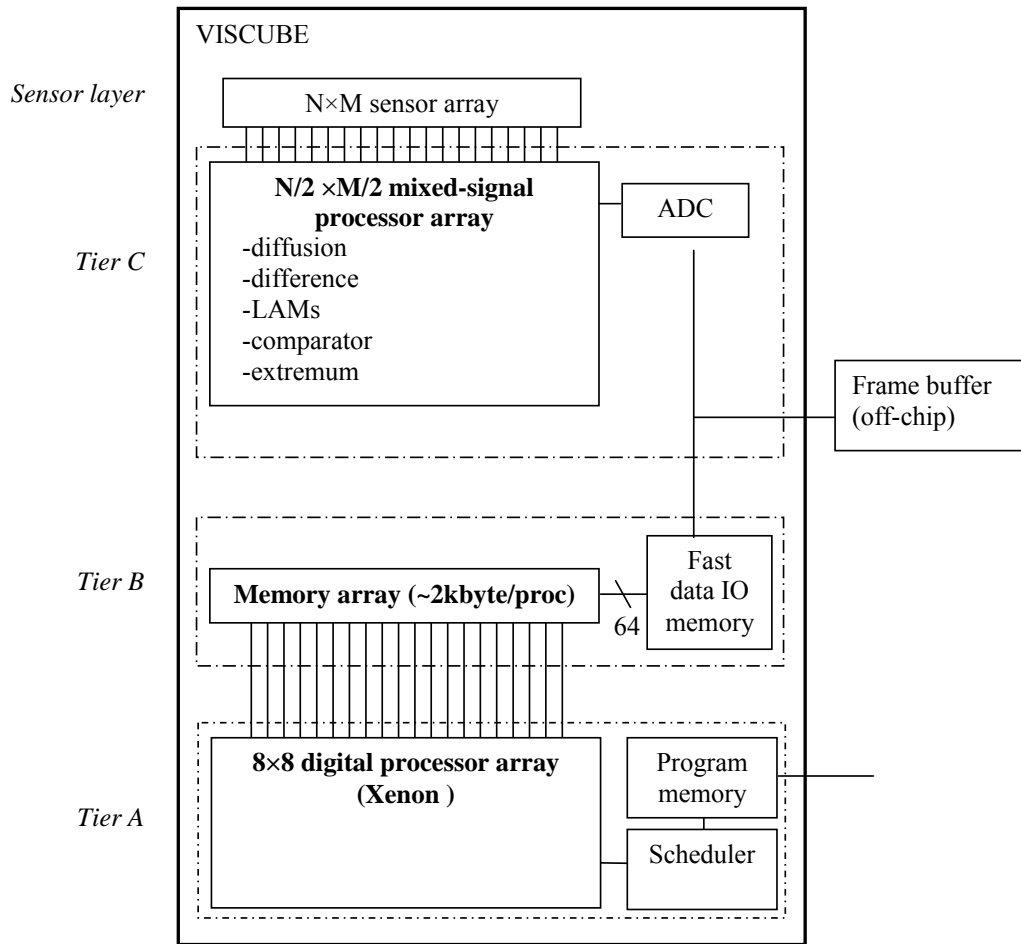


Figure 38 VISCUBE chip architecture

3.5.1.2 Mixed-signal processor layer

Tier C contains a fully-programmable smart image processor cell array (derivative of the Q-Eye [67] Section 4.1.4) with embedded sensor interface. Its resolution will be half of the sensor array in both dimensions. The sensor array is topographically mapped to the processor array on a way that on the top of each processor cells, there are 4 sensor cells. This is called pitch matched design. In this way, each processor cell collects photocurrents from the four pixels which are physically above it. The captured pixel values are stored in analog memories, hence data conversion is not needed.

The block diagram of Tier C is shown in Figure 39. Besides the mixed-signal processor array, it contains the control unit, and an AD converter. Each of the cells contains a sensor interface unit, an analog arithmetic unit, a local IO unit, a diffusion network unit, an analog memory block, a comparator, a logic memory block, and an external IO unit.

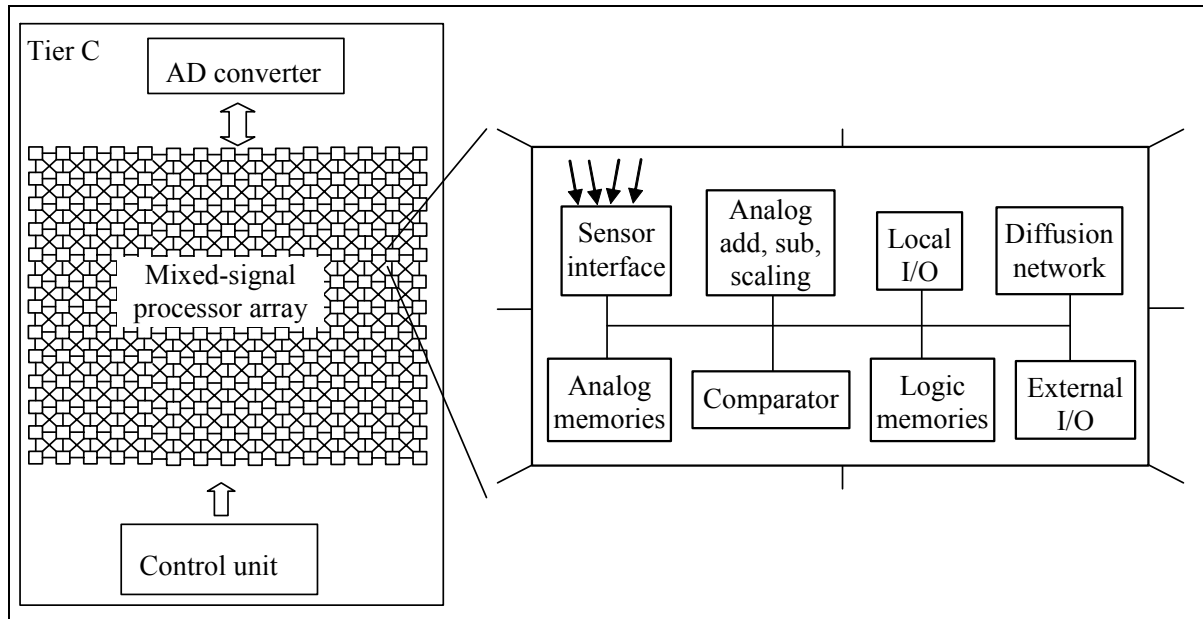


Figure 39 Block diagram of Tier C

The mixed-signal processor array can perform the following operations:

- Image acquisition;
- Storing multiple grayscale or binary images (sensor readouts, subresults, and final results);
- Adding, subtracting, scaling grayscale images;
- Shifting grayscale or binary images;
- Applying diffusion operator by using an embedded resistive grid;
- Comparing grayscale images with each other or with a constant value.

The image acquisition can be performed with the same time as the other operations. Similar to a classic CNN circuit, the processor array operates in a single instruction multiple data (SIMD) mode. However, the operations are more atomic here. While in a CNN, the basic operators are feed-forward or feedback convolutions implemented by local or global spatial-temporal analog transients, here a convolution is put together from a sequence of shifts, scalings, and additions or subtractions, as we would do it on a microprocessor. Naturally each operation is executed on the whole image in parallel.

As an example of efficient operation, here we show how this computer array can calculate the high-pass, band-pass, and low-pass components of an image, and how it can extract local maxima. We consider that the image is in local analog memory zero (LAM0).

```

Diffuse_image →LAM1;
Diffuse_image LAM1 →LAM2;           // LAM2: low-pass component
Subtract_image LAM1 LAM2 →LAM3;      // LAM3: band-pass component
Subtract_image LAM0, LAM1 →LAM4;     // LAM4: high-pass component
Threshold_image LAM0, LAM1 →LLM0     // LLM0: local maximum places

```

This operation sequence takes roughly 30 microseconds for the whole image. The high-pass, band-pass, and the low-pass components are the bases of multi-scale analysis, while the local maximum¹ indicates the irregularities of the image in high contrast means. These locations are good candidates for further foveal processing. The local maximum places are stored in a binary (one bit/pixel) image, which is called local logic image (LLM).

The mixed-signal processor array can handle binary or analog images. The binary images are read out directly in one bit/pixel form, while the analog (grayscale) images needs AD conversion. To be able to increase the IO speed, and support the foveal imaging concept, both the binary and the grayscale images can be read out in arbitrary sized windows. To support multi-scale analysis, the windows can be down-sampled, which further reduce the IO time requirements.

3.5.1.3 Digital processor array layer

The digital processor array layer is a derivative of the Xenon processor [12][6]. It is used for foveal processing. Different applications require different window (fovea) sizes. For example, when we need to find the exact matching position of a large number of feature points, quick calculations in 8x8 or 16x16 windows are required. However, when deep feature analysis of a navigating object is required, we need to process 32x32 or 64x64 sub-images, depending on the size of the object. Therefore, we designed the digital processor layer to be able to efficiently handle these different frame-sizes. Hence, the foveal processor array is not scalable. If we scale up the design, it is worth to add extra arithmetic units or memory to the foveal processor array, to increase its speed, but it does not make sense to significantly increase the fovea size.

The basic constructing element of our digital processor array architecture is the cell (Figure 40). The cells are locally interconnected, thus the processors in each cell can read the memory of their direct neighbors. There are boundary cells, which are relying data to handle different boundary conditions.

¹ The local maximum is approximated as follows. The low-pass component of the image is subtracted from the original image. Since the low-pass component approximates the local average of the image in every locations, the difference of the original image and its low-pass component will be large, where the original image is significantly larger than its local average. These locations are the local maxima. The method can find the local peaks, but it may mark some surrounding pixels too.

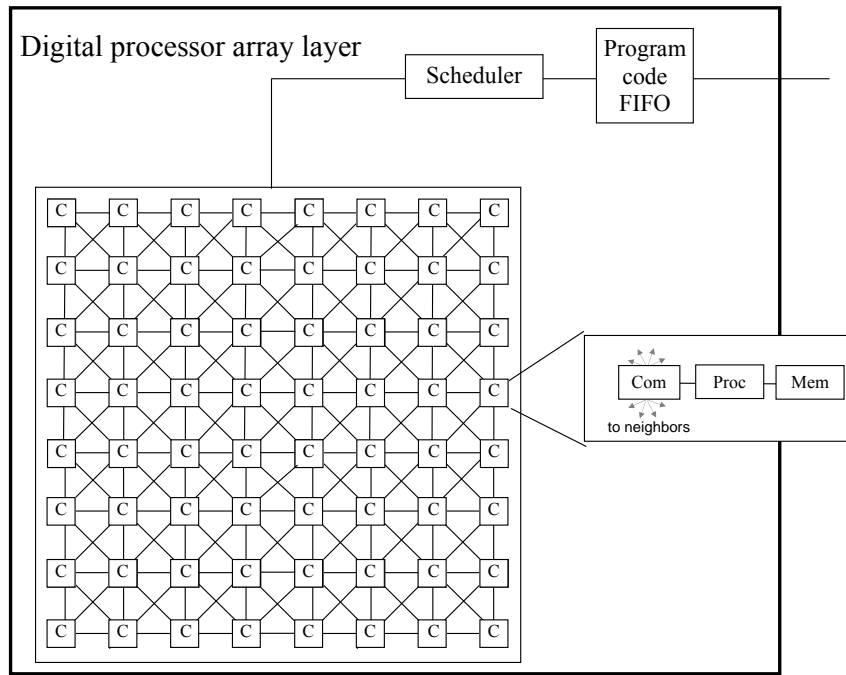


Figure 40 The architecture of the digital processor array. It is constructed of 64 cells. Each cell handles 4, 16, or 64 pixels.

Each cell (Figure 41) contains an arithmetic processor unit, a morphologic processor unit, data memory, internal and external communication unit. The arithmetic unit contains an 8 bit multiple-add processor with a 24 bit accumulator, and 8 pieces of 8 bit registers. This makes possible to perform either 8 or 16 bit precision calculations.

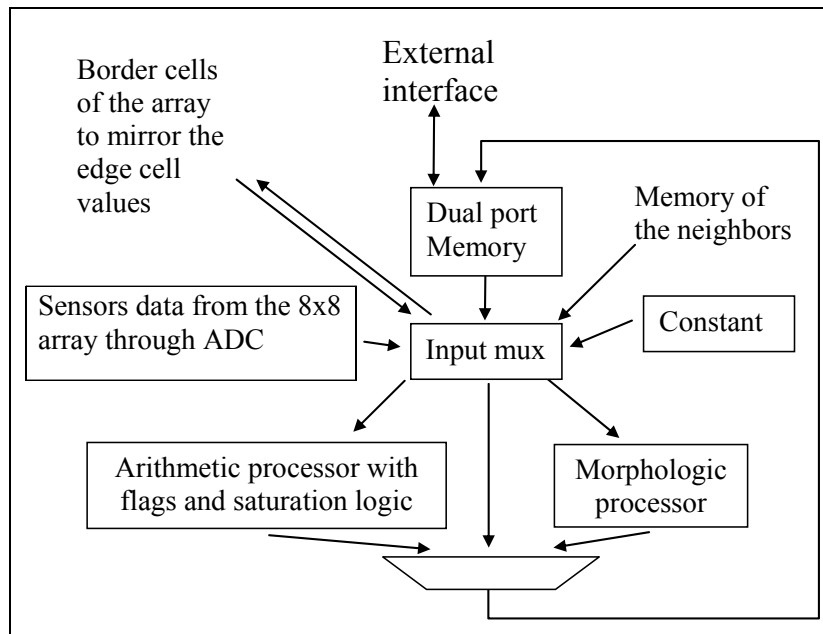


Figure 41 The cell architecture of the digital processor array layer

The morphology unit supports the processing of black-and-white images. It contains 8 pieces of single bit morphology processor, for parallel calculation of local or spatial logic operations, like erosion, dilation, opening, closing, hit and miss operations, etc.

Each cell is prepared to process maximum 8x8 sized subimages (64 pixels). From the simulations, it turned out, that the storage of 8 subimages is satisfactory. Hence, each processor cell requires minimum 512 bytes of local memory. The processor cells are connected with their neighbors on a way that they can read each others memory is a single cycle.

The next sample code shows how a convolution is calculated in the digital processor layer. This sample code calculates one pixel/processor-cell, and it is repeated for each pixel as many times as many pixels are handled by the processor cell. This repetition is done by the control unit of the digital processor layer automatically. [arg 1] is the input memory location and [arg2] is the output memory location. The coordinates after the memory address [arg1] points to the neighborhood of the processed pixel. If the pixel is on the boundary of the subunit, some of the neighboring pixel data comes from the neighboring cell. Since the memory of the neighboring cell can be accessed without any bottleneck, it does not require special code. Coeff1 to coeff9 are the coefficients of the convolution.

```
mov.mem.boundary    Mem[ arg1 ];

mul.nbr.const       Mem[ arg1 ][-1,-1], coeff1;
macc.nbr.const      Mem[ arg1 ][0,-1], coeff2;
macc.nbr.const      Mem[ arg1 ][1,-1], coeff3;

macc.nbr.const      Mem[ arg1 ][-1,1], coeff4;
macc.nbr.const      Mem[ arg1 ][0,1], coeff5;
macc.nbr.const      Mem[ arg1 ][1,1], coeff6;

macc.nbr.const      Mem[ arg1 ][-1,0], coeff7;
macc.nbr.const      Mem[ arg1 ][1,0] coeff8;
macc.nbr.const      Mem[ arg1 ][0,0], coeff9;

acc.shl;
sat16.mem           Mem[ arg2 ];
```

The cells do not have local program memory. The program is coming from a scheduler: each processor receives the same command, parameters and attributes in each time step, which makes it a SIMD processor array architecture. The individual processing cells are maskable, which means that content-dependent masks may enable or disable the execution of a certain image processing operation in any pixel locations. This masking can make the process locally adaptive.

The instruction set of the digital processor array contains five groups:

- Initialization instructions
- Data transfer instructions
- Arithmetic instructions
- Logic instructions
- Comparison instructions

The Initialization instructions are needed to clear or set the accumulator, the boundary condition registers, the masks, and other registers of the cells.

The data transfer instructions are used to transfer data between the internal registers and the memory. The cells can access the memory of their direct neighbors too. In this way, a processor can access either its own memory or any of its direct neighbor memory, as it was already mentioned.

The arithmetic operation set contains addition, subtraction, multiplication, multiple-add operation, and shift. These operators set the flags of the arithmetic units. These flags can be used in the next instruction as conditions.

The comparison instructions are introduced to calculate the relation between two scalars. These operators can be used for statistical filter implementations.

Using these instructions, we can efficiently implement the basic image processing functions (convolution, statistical filters, gradient, grayscale and binary mathematical morphology, etc) on the processor array.

3.5.2 Data communication, conversion, scaling

The fast and efficient internal and external communication is one of the key essences of the architecture. According to the specification, the system can operate between 1000 and 5,000 FPS. This fast operation requires not only fast processing, but fast and efficient communication too.

Communication in the system mostly means transferring entire images, or scaled images, or windows. Figure 42 shows both the data communication and the control channels. Image data is transferred via a 32 bit wide bus between an accompanying RISC processor and the digital processor array.

The analog image data stored in the distributed memories of the mixed-signal processor array can be accessed through an AD converter. This provides a random access to the analog memories of the mixed-signal array. A data organizer unit is used to pack 1 bit or 8 bit data to 32 bit words. This means that it collects 4 consecutive grayscale pixels, or 32 consecutive binary pixels, and put them into one 32 bit long word.

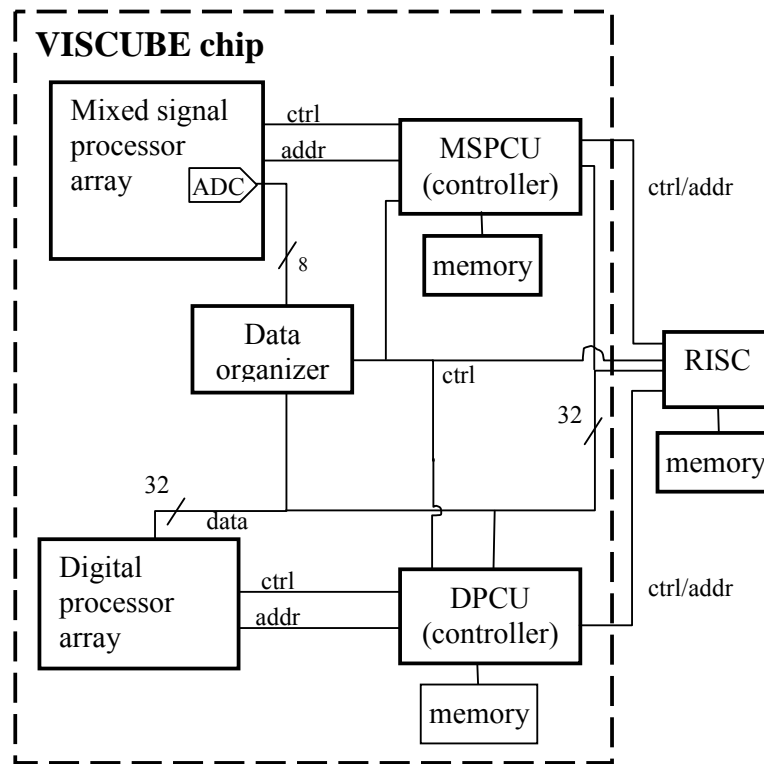


Figure 42 The communication channels of the VISCUBE chip

3.5.3 Operation, control and synchronization

The VISCUBE processor needs an accompanying processor to serve as a control, communication, and final decision making device. This can be practically a *reduced instruction set* processor (RISC), because it should be able to operate and communicate on high speed, however, it does not need to perform computationally demanding calculations.

In the final application environment, this RISC executes the main program of the system. It is responsible for initializing subroutines on the individual processor array layers, and image data communication among the three processing units. The RISC processor continuously evaluates the captured and preprocessed image flows arriving from the mixed-signal layer, and decide which parts (windows) of the input image requires more detailed analysis, and orders digital processor array to perform it. It is also responsible to switch between algorithms (subroutines), or modify the process arguments according to the input image contents.

Each of the processor array layers needs control processors. These two control processors store a couple of routines, and execute them as it is ordered by the RISC. These routines may contain not just computational instructions, but also conditional program flow control and synchronization instructions, as well. The condition may depend either on internal variables, or on the input image contents, or on external events.

The synchronization of the different processors is done through flags. There are 16 flags defined, each of them represent the state of a condition. Each processor can set or reset any of

the flags. The processes can be started conditionally and unconditionally. Conditionally started processes cannot start until the associated condition is true.

3.5.4 Target algorithms: registration

The primary application of the VISCUBE is airborne visual navigation. This application is based on segmentation. However, segmentation of an image provided by a camera on a moving platform requires registration. Image registration means to find and calculate the series of projective transformations caused by the moving camera. One of the target algorithm of the VISCUBE is the image registration (Figure 43).

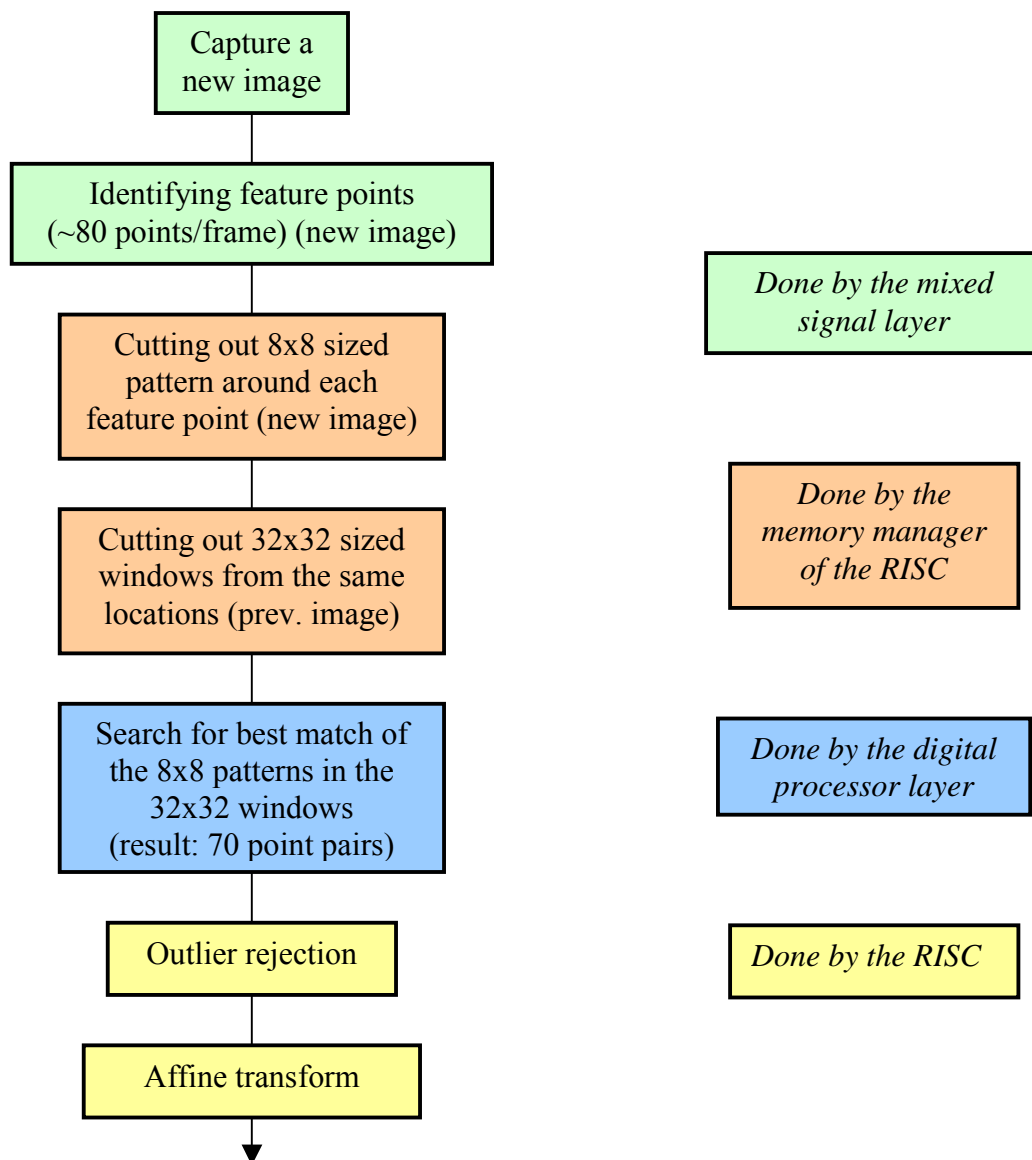


Figure 43 Flow-chart of a typical image registration algorithm

First, it requires the identification of the characteristic points of the image. This is typically done by the Harris corner detector algorithm, however, that cannot be implemented

in the mixed-signal processor. Therefore, it is replaced with local maximum/minimum point identification in different scales.

The expected output of the first step of the registration algorithm is a set of about 80 feature points. If the algorithm can provide a rough estimation for their placement that could greatly reduced the required processing power in the searching step.

The digital processor layer does the search of the matching positions of the 8x8 patterns within the 32x32 search windows. Last steps of the algorithm are the outlier rejection and the completion of the affine transform, which are done by the RISC processor.

3.6 Conclusions

Virtual processor arrays were introduced for both early and post image processing tasks. They enable the usage of the topographic processor arrays in video or even in megapixel application without significant efficiency drop. The utilization of the introduced virtual processing arrays differs. The elongated processor array architecture was covered with a Hungarian and an international PCT patent application [4][5]. Though the architecture has never been implemented in this form, it paved the road to a subsequent digital version (described in Section 4.1.2), which has been implemented on FPGA and currently used for video analytics in security applications [22] in the industry.

The ASIC implementation of the CASTLE architecture was started in 2002 in the framework of an OMFB (Hungarian Research Found) project. The architecture of the CASTLE chip motivated the Falcon architecture [50], which is also a many-core digital CNN-UM emulator processor implemented on FPGA.

In 2003, when the Bi-i camera [8][9][19] was completed, it was the fastest camera in industry. It has received the Product of the Year Award in the industrial Vision Fair in Stuttgart, Germany in 2003. A few dozens of Bi-i cameras were built and sold to academic-, university-, and industrial research laboratories all over the world. Its novel multi-scale, multi-fovea approach attracted a research grant from NASA Jet Propulsion Laboratory to Hungary, to investigate whether it is possible to use a miniaturized version of the Bi-i as a visual navigation and reconnaissance device of a UAV, navigating autonomously in Mars' atmosphere, seeking for water carved surface formations.

The Viscube chip is the miniaturized version of the Bi-i in some sense. It is designed to be a visual navigation and reconnaissance device on small UAV platform. Its novelty is that it is the first device which combines two topographic processor arrays: a medium resolution mixed-signal one, and a small resolution digital foveal one on a single chip. Its integration technology is novel also, because it is implemented as a test project of the experimental 3D integration technology.

4 Low-power processor array design strategy for solving computationally intensive 2D topographic problems

Cellular Neural/nonlinear Networks (CNN) were invented in 1988 [24]. This new field attracted well beyond hundred researchers in the next two decades, called nowadays the CNN community. They focused on three main areas: the theory, the implementation issues, and the application possibilities. In the implementation area, the first 10 years yielded more than a dozen CNN chips made by only a few designers. Some of them followed the original CNN architecture [39], others made slight modifications, such as the full signal range model [43] [45], or discrete time CNN (DTCNN) [40], or skipped the dynamics, and made dense threshold logic in the black-and-white domain [41] only. All of these chips had cellular architecture, and implemented the programmable **A** and/or the **B** template matrices of the CNN Universal Machine [27] [21]

In the second decade, this community slightly shifted the focus of chip implementation. Rather than implementing classic CNN chips with **A** and **B** template matrices, the new target became the efficient implementation of neighborhood processing. Some of these architectures were topographic with different pixel/processor ratios, others were non-topographic. Some implementations used analog processors and memories, others digital ones. Certainly, the different architectures had different advantages and drawbacks. One of the goals is to compare these architectures and the actual chip implementations themselves. This attempt is not trivial, because their parameter gamut and operation modes are rather different. To solve this problem, we have categorized the most important 2D wave type operations and examined their implementation methods and efficiency on these architectures.

In this study, I have compared the following five architectures, of which the first one is used as the reference of comparison.

1. DSP-memory architecture (in particular DaVinci processors from TI [59])
2. Pipe-line architecture (CASTLE [3][2], Falcon [50])
3. Coarse-grain cellular parallel architecture (Xenon [13]);
4. Fine-grain fully parallel cellular architecture with discrete time processing (SCAMP [49], Q-Eye [67]);
5. Fine-grain fully parallel cellular architecture with continuous time processing (ACE-16k [42], ACLA [46][47]).

Based on the result of this analysis, I have calculated the major implementation parameters of the different operation classes for every architectures. These parameter are the maximal resolution, frame-rate, pixel clock, and computational demand, the minimal latency,

and the flow-chart topology. Having these constraints, the optimal architecture can be selected to a given algorithm. The architecture selection method is described.

The analysis of the 2D wave type operators on different many-core architectures and the optimal architecture selection method are my work. Parts of these results were described in [16], and a more detailed journal paper is under publication [17].

The chapter starts with the brief description of the different architectures (Section 4.1), which is followed by the categorization of the 2D operators and their implementation methods on them (Section 4.2). Then the major parameters of the implementations are compared (Section 4.3). Finally, in Section 4.4 the optimal architecture selection method is introduced.

4.1 Architecture descriptions

In this section, we describe the architectures examined using the basic spatial grayscale and binary functions (convolution, erosion) of non-propagating type.

4.1.1 Classic DSP-memory architecture

Here we assume a 32 bit DSP architecture with cache memory large enough to store the required number of images and the program internally. In this way, we have to practically estimate/measure the required DSP operations. Most of the modern DSPs have numerous MACs and ALUs. To avoid comparing these DSP architectures, which would lead too far from our original topic, we use the DaVinci video processing DSP by Texas Instrument, as a reference.

We use 3×3 convolution as a measure of grayscale performance. The data requirements of the calculation are 19 bytes (9 pixels, 9 kernel values, result), however, many of these data can be stored in registers, hence, only as an average of a four-data access (3 inputs, because the 6 other ones had already been accessed in the previous pixel position, and one output) is needed for each convolution. From computational point of view, it needs 9 multiple-add (MAC) operations. It is very typical that the 32 bit MACs in a DSP can be split into four 8 bit MACs, and other auxiliary ALUs help loading the data to the registers in time. Measurement shows that, for example, the Texas DaVinci family with the TMS320C64x core needs only about 1.5 clock cycles to complete a 3×3 convolution.

The operands of the binary operations are stored in 1 bit/pixel format, which means that each 32bit word represents a 32×1 segment of an image. Since the DSP's ALU is a 32 bit long unit, it can handle 32 binary pixels in a single clock cycle. As an example, we examine how a 3×3 square shaped erosion operation is executed. In this case erosion is a nine input OR operation where the inputs are the binary pixels values within the 3×3 neighborhood. Since the ALU of the DSP does not contain 9 input OR gate, it is executed sequentially on 32 an

entire 32×1 segment of the image. The algorithm is simple: the DSP has to prepare the 9 different operands, and apply bit-wise OR operations on them.

Figure 44 shows the generation method of the first three operands. In the figure a 32×3 segment of a binary image is shown (9 times), as it is represented in the DSP memory. Some fractions of horizontal neighboring segments are also shown. The first operand can be calculated by shifting the upper line with one bit position to the left and filling in the empty MSB with the LSB of the word from its right neighbor. The second operand is the un-shifted upper line. The position and the preparation of the remaining operands are also shown in Figure 44a.

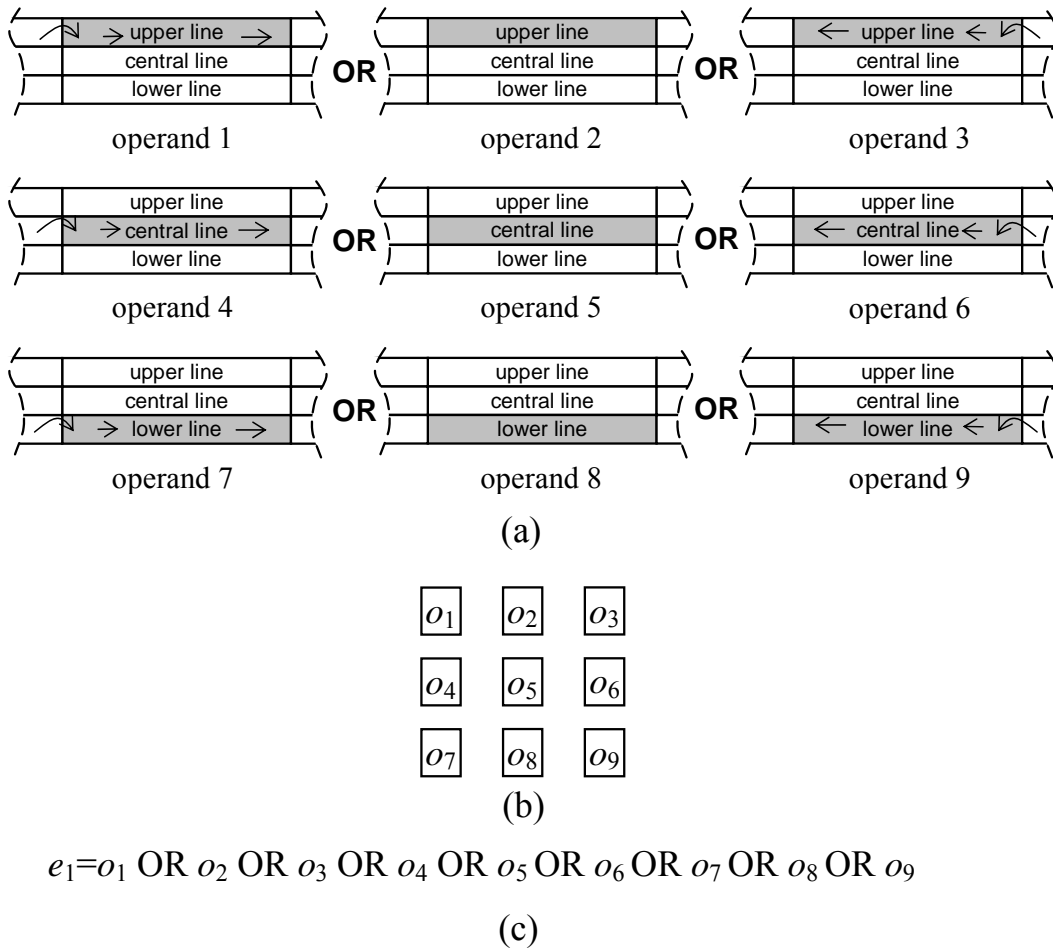


Figure 44. Illustration of the binary erosion operation on a DSP. (a) shows the 9 pieces of 32×1 segments of the image (operands), as the DSP uses them. The operands are the shaded segments. The arrows indicate shifting of the segments. To make it clearer, consider a 3×3 neighborhood as it is shown in (b). For one pixel, the form of the erosion calculation is shown in (c). o_1, o_2, \dots, o_9 are the operands. The DSP does the same, but on 32 pixels parallel.

This means that we had to apply 10 memory accesses, 6 shifts, 6 replacements, and 8 OR operations to execute a binary morphological operation for 32 pixels. Due to the multiple

cores and the internal parallelism, the Texas DaVinci spends 0.5 clock cycles with the calculation of one pixel.

In the low power low cost embedded DSP technology the trend is to further increase the clock frequency, but most probably, not higher than 1 GHz, otherwise, the power budget cannot be kept. Moreover, the drawback of these DSPs is that their cache memory is too small, which cannot be significantly increased without significant cost rise. The only way to significantly increase the speed is to implement a larger number of processors, however, that requires a new way of algorithmic thinking, and software tools.

The DSP-memory architecture is the most versatile from the point of views of both in functionality and programmability. It is easy to program, and there is no limit on the size of the processed images, though it is important to mention that in case of an operation is executed on an image stored in the external memory, its execution time is increasing roughly with an order of magnitude. Though the DSP-memory architecture is considered to be very slow, as it is shown later, it outperforms even the processor arrays in some operations. In QVGA frame size, it can solve quite complex tasks, such as video analytics in security applications on video rate [71]. Its power consumption is in the 1-3W range. Relatively small systems can be built by using this architecture. The typical chip count is around 16 (DSP, memory, flash, clock, glue logic, sensor, 3 near sensor components, 3 communication components, 4 power components), while this can be reduced to the half in a very basic system configuration.

4.1.2 Pipe-line architectures

We have already considered pipe-line processor arrays in Section 3.2 and 3.3, which were specially designed for CNN calculation. Here, a general digital pipe-line architecture with one processor core per image line arrangement will be briefly introduced. The basic idea of this pipe-line architecture is to process the images line-by-line, and to minimize both the internal memory capacity and the external IO requirements. Most of the early image processing operations are based on 3×3 neighborhood processing, hence 9 image data are needed to calculate each new pixel value. However, these 9 data would require very high data throughput from the device. As we will see, this requirement can be significantly reduced by applying a smart feeder arrangement.

Figure 45 shows the basic building blocks of the pipe-line architecture. It contains two parts, the memory (feeder) and the neighborhood processor. Both the feeder and the neighborhood processor can be configured 8 or 1 bit/pixel wide, depending on whether the unit is used for grayscale or binary image processing. The feeder contains, typically, two consecutive whole rows and a row fraction of the image. Moreover, it optionally contains two more rows of the mask image, depending on the input requirements of the implemented neighborhood operator. In each pixel clock period, the feeder provides 9 pixel values for the

neighborhood processor and the mask value optionally if the operation requires it. The neighborhood processor can perform convolution, rank order filtering, or other linear or nonlinear spatial filtering on the image segment in each pixel clock period. Some of these operators (e.g., *hole finder*, or a *CNN* emulation with A and B templates) require two input images. The second input image is stored in the mask. The outputs of the unit are the resulting and, optionally, the input and the mask images. Note that the unit receives and releases synchronized pixels flows sequentially. This enables to cascade multiple pieces of the described units. The cascaded units forms a chain. In such a chain, only the first and the last units require external data communications, the rest of them receives data from the previous member of the chain and releases the output towards the next one.

An advantageous implementation of the row storage is the application of FIFO memories, where the first three positions are tapped to be able to provide input data for the neighborhood processor. The last positions of rows are connected to the first position of the next row (Figure 45). In this way, pixels in the upper rows are automatically marching down to the lower rows.

The neighborhood processor is of special purpose, which can implement one or a few different kinds of operators with various attributes and parameter. They can implement convolution, rank-order filters, grayscale or binary morphological operations, or any local image processing functions (e.g. Harris corner detection, Laplace operator, gradient calculation, etc.). In architectures CASTLE [3][2] and Falcon [50], e.g., the processors are dedicated to convolution processing where the template values are the attributes. The pixel clock is matched with that of the applied sensor. In case of a 1 megapixel frame at video rate (30 FPS), the pixel clock is about 30 MHz (depending on the readout protocol). This means that all parts of the unit should be able to operate minimum on this clock frequency. In some cases the neighborhood processor operates on an integer multiplication of this frequency, because it might need multiple clock cycles to complete a complex calculation, such as a 3×3 convolution. Considering ASIC or FPGA implementations, clock frequency between 100-300 MHz is a feasible target for the neighborhood processors within tolerable power budget.

The multi-core pipe-line architecture is built up from a sequence of such processors. The processor arrangement follows the flow-chart of the algorithm. In case of multiple iterations of the same operation, we need to apply as many processor kernels, as many iterations we need. This easily ends up in using a few dozens of kernels. Fortunately, these kernels, especially in the black-and-white domain, are relatively inexpensive, either on silicon, or in FPGA.

Depending on the application, the data-flow may contain either sequential segments or parallel branches. It is important to emphasize, however, that the frame scanning direction

cannot be changed, unless the whole frame is buffered, which can be done in external memory only. Moreover, the frame buffering introduces relatively long (dozens of millisecond) additional latency.

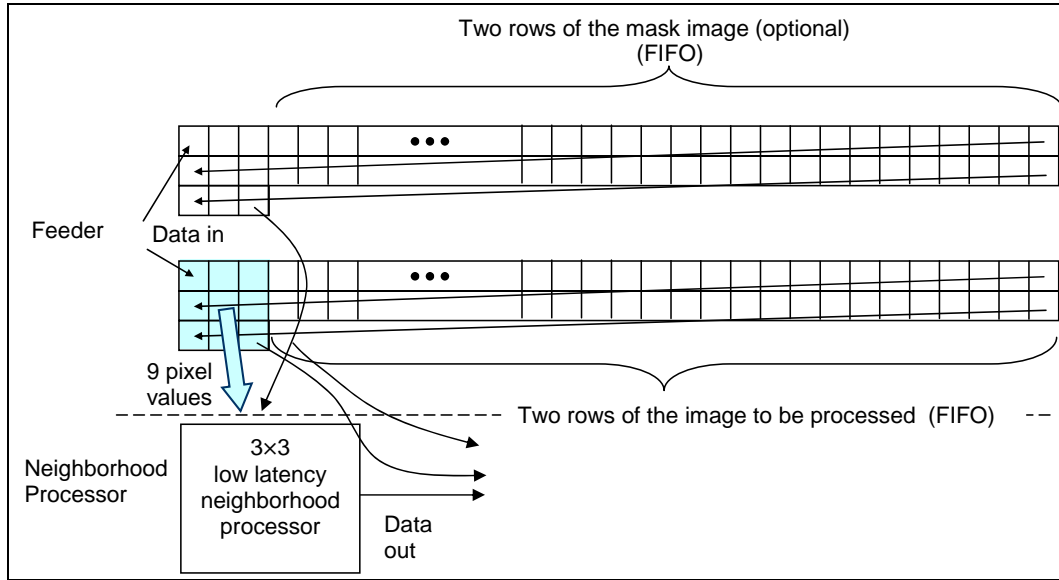


Figure 45. One processor and its memory arrangement in the pipe-line architecture.

For capability analysis, here we use the Spartan 3ADSP FPGA (XC3SD3400A) from Xilinx [64] as a reference, because this low-cost, medium performance FPGA was designed especially for embedded image processing. It is possible to implement roughly 120 grayscale processors within this chip, as long as the image row length is below 512, or 60 processors, when the row length is between 512 and 1024.

4.1.3 Coarse-grain cellular parallel architectures

We have already discussed a coarse-grain cellular architecture in Section 3.5.1.3 as the digital foveal processor of the Viscube architecture. In that case, the coarse-grain architecture received input from a fine-grain mixed signal layer. As a contrast, the Xenon [13] architecture (briefly shown here) is equipped with an embedded photosensor array.

The coarse-grain architecture is a truly locally interconnected 2D cellular processor arrangement, as opposed to the pipe-line one. A specific feature of the coarse-grain parallel architectures is that each processor cell is topographically assigned to a number of pixels (e.g., an 8×8 segment of the image), rather than to a single pixel only. Each cell contains a processor and some memory, which is large enough to store few bytes for each pixel of the allocated image segment. Exploiting the advantage of the topographic arrangement, the cells can be equipped with photo sensors enabling to implement a single chip sensor-processor device. However, to make this sensor sensitive enough, which is the key in high frame-rate applications, and to keep the pixel density of the array high, at the same time, certain vertical integration techniques are needed for photosensor integration.

In the coarse-grain architectures, each processor serves a larger number of pixels, hence we have to use more powerful processors, than in the one-pixel per processor architectures. Moreover, the processors have to switch between serving pixels frequently, hence more flexibility is needed that an analog processor can provide. Therefore, it is more advantageous to implement 8 bit digital processors, while the analog approach is more natural in the one pixel per processor (fine-grain) architectures. (See the next subsection.)

As it can be seen in Figure 46, Xenon chip is constructed of an 8×8 , locally interconnected cell arrangement. Each cell contains a sub-array of 8×8 photosensors; an analog multiplexer; an 8 bit AD converter; an 8 bit processor with 512 bytes of memory; and a communication unit of local and global connections. The processor can handle images in 1, 8, and 16 bit/pixel representations, however, it is optimized for 1 and 8 bit/pixel operations. Each processor can execute addition, subtraction, multiplication, multiply-add operations, comparison, in a single clock cycle on 8 bit/pixel data. It can also perform 8 logic operations on 1 bit/pixel data in packed-operation mode in a single cycle. Therefore, in binary mode, one line of the 8×8 sub-array is processed jointly, similarly to the way we have seen in the DSP. However, the Xenon chip supports the data shifting and swapping from hardware, which means that the operation sequence, what we have seen in Figure 44 takes 9 clock cycles only. (The swapping and the accessing the memory of the neighbors do not need extra clock cycles.) Besides, the local processor core functions, Xenon can also perform a global OR function. The processors in the array are driven in a single instruction multiple data (SIMD) mode.

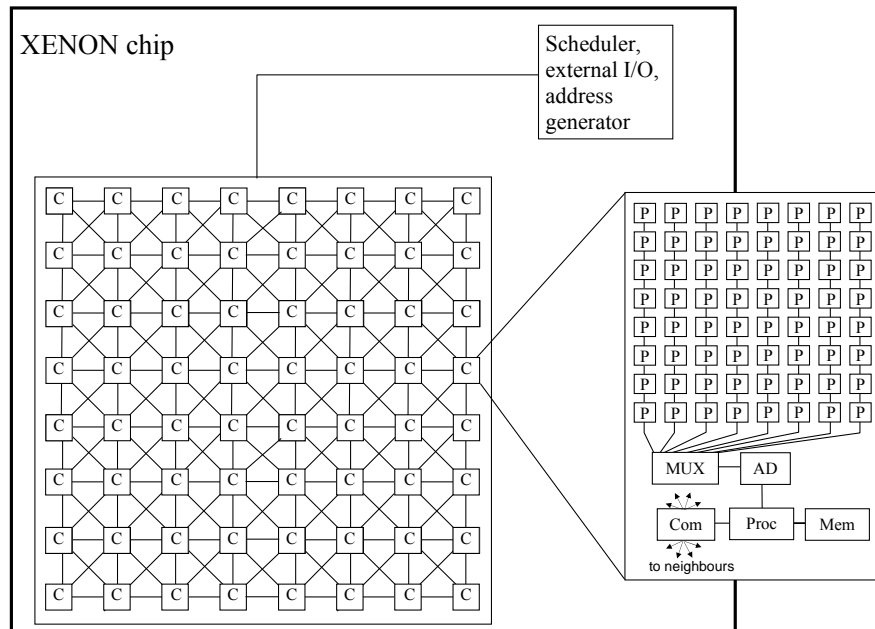


Figure 46. Xenon is a 64 core coarse-grain cellular parallel architecture (C stands for processor cores, while P represents pixels).

Xenon is implemented on a 5x5mm silicon die with 0.18 micron technology. The clock cycle can go up to 100MHz. The layout is synthesized, hence the resulting 75micron equivalent pitch is far from being optimal. It is estimated that through aggressive optimization it could be reduced to 40 micron (assuming a bump bonded sensor layer), which would make almost double the resolution achievable on the same silicon area. The power consumption of the existing implementation is under 20mW.

4.1.4 Fine-grain fully parallel cellular architectures with discrete time processing

The fine-grain, fully parallel architectures are based on rectangular processor grid arrangements where the 2D data (images) are topographically assigned to the processors. The key feature here is that there is a one-to-one correspondence between the pixels and the processors. This certainly means that at the same time the composing processors can be simpler and less powerful, than in the previous, coarse-grain case. Therefore, fully parallel architectures are typically implemented in analog domain, though bit-sliced digital approach is also feasible. The introduced mixed-signal processor array of the Viscube (Section 3.5.1.2) is one example of this type of processor architectures.

In the discussed cases, the discrete time processing type fully parallel architectures are equipped with a general purpose, analog processor, and an optical sensor in each cell. These sensor-processors can handle two types of data (image) representations: grayscale and binary. The instruction set of these processors include addition, subtraction, scaling (with a few discrete factors only), comparison, thresholding, and logic operations. Since it is a discrete time architecture, the processing is clocked. Each operation takes 1-4 clock cycles. The individual cells can be masked. Basic spatial operations, such as convolution, median filtering, or erosion, can be put together as sequences of these elementary processor operations. In this way the clock cycle counts of a convolution, a rank order filtering, or a morphologic filter are between 20 and 40 depending on the number of weighting coefficients.

It is important to note that in case of the discrete time architectures (both coarse- and fine-grain), the operation set is more elementary (lower level) than on the continuous time cores (see the next section). While in the continuous time case (CNN like processors) the elementary operations are templates (convolution, or feedback convolution) [26][27], in the discrete time case, the processing elements can be viewed as RISC (reduced instruction set) processor cores with addition, subtraction, scaling, shift, comparison, and logic operations. When a full convolution is to be executed, the continuous time architectures are more efficient. While in the case of operations when both architectures apply a sequence of elementary instructions in an iterative manner (e.g., rank order filters), the RISC is the superior, because its elementary operators are more versatile more accurate, and faster.

The internal analog data representation has both architectural and functional advantages. From architectural point of view, the most important feature is that no AD converter is needed

on the cell level, because the sensed optical image can be directly saved in the analog memories, leading to significant silicon space savings. Moreover, the analog memories require smaller silicon area than the equivalent digital counterparts. From the functional point of view, the topographic analog and logic data representations make the implementation of efficient diffusion, averaging, and global OR networks possible.

The drawback of the internal analog data representation and processing is the signal degradation during operation or over time. According to experience, accuracy degradation was more significant in the old ACE16k design [42] than in the recent Q-Eye [67] or SCAMP [49] ones. While in the former case 3-5 grayscale operations led to significant degradations, in the latter ones even 10-20 grayscale operations can conserve the original image features. This makes it possible to implement complex nonlinear image processing functions (e.g., rank order filter) on discrete time architectures, while it is practically impossible on the continuous ones (ACE16k).

The two representatives of discrete time solutions, SCAMP and Q-Eye, are slightly similar in design. The SCAMP chip was fabricated by using 0.35 micron technology. The cell array size is 128×128 . The cell size is 50×50 micron, and the maximum power consumption is about 200mW at 1.25MHz clock rate. The array of Q-Eye chip has 144×176 cells. It was fabricated on 0.18 micron technology. The cell size is about 30×30 micron. Its speed and power consumption range is similar to that of the SCAMP chip. Both SCAMP and Q-Eye chips are equipped with single-step mean, diffusion, and global OR calculator circuits. Q-Eye chip also provides hardware support to single-step binary 3×3 morphologic operations.

4.1.5 Fine-grain fully parallel cellular architecture with continuous time processing

Fully parallel cellular continuous time architectures are based on arrays of spatially interconnected dynamic asynchronous processor cells. Naturally, these architectures exhibit fine-grain parallelism, to be able to perform continuous time spatial waves physically in the continuous value electronic domain. Since these are very carefully optimized, special purpose circuits, they are super-efficient in computations they were designed to. We have to emphasize, however, that they are not general purpose image processing devices. Here we mainly focus on two designs. Both of them can generate continuous time spatial-temporal propagating waves in a programmable way. While the output of the first one (ACE-16k [42]) can be in the grayscale domain, the output of the second one (ACLA [46][47]) is always in the binary domain.

The ACE-16k [42] is a classical CNN Universal Machine type architecture equipped with feedback and feed-forward template matrices [27], sigmoid type output characteristics, dynamically changing state, optical input, local (cell level) analog and logic memories, local logic, diffusion and averaging network. It can perform full-signal range type CNN operations [35]. Therefore, it can be used in retina simulations or other spatial-temporal dynamical

system emulations, as well. Its typical feed-forward convolution execution time is in the 5-8 microsecond range, while the wave propagation speed from cell-to-cell is up to 1 microsecond. Though its internal memories, easily re-programmable convolution matrices, logic operations, and conditional execution options make it attractive to use as a general purpose high-performance sensor-processor chip for the first sight, its limited accuracy, large silicon area occupation ($\sim 80 \times 80$ micron/cell on 0.35 micron 1P5M STM technology), and high power consumption (4-5 Watts) prevent the immediate usage in various vision application areas.

The other architecture in this category is the Asynchronous Cellular Logic Array (ACLA) [46], [47]. This architecture is based on spatially interconnected logic gates with some cell level asynchronous controlling mechanisms, which allow ultra high-speed spatial binary wave propagation only. Typical binary functionalities implemented on this network are: *trigger wave*, *reconstruction*, *hole finder*, *shadow*, etc. Assuming more sophisticated control mechanism on the cell level, it can even perform *skeletonization* or *centroid* calculations. Their implementation is based on a few minimal size logic transistors, which makes them hyper-fast, extremely small, and power-efficient. They can reach 500 ps/cell wave propagation speed, with 0.2mW power consumption for a 128×128 sized array. Their very small area requirement (16×8 micron/cell on 0.35 micron 3M1P AMS technology) makes them a good choice to be implemented as a co-processor in any fine-grain array processor architecture.

4.2 Implementation and efficiency analysis of various operators

Based on the implementation methods, in this section, we introduce a new 2D operator categorization. Then, the implementation methods on different architectures are described and analyzed from the efficiency aspect.

Here we examine only the 2D single-step neighborhood operators, and the 2D, neighborhood based wave type operators. The more complex, but still local operators (such as Canny edge detector) can be built up by using these primitives, while other operators (such as Hough or Fourier transform) require global processing, which is not supported by these architectures.

4.2.1 Categorization of 2D operators

The calculation methods of different 2D operators, due to their different spatial-temporal dynamics, require different computational approaches. The categorization (Figure 47) was done according to their implementation methods on different architectures. It is important to emphasize that we categorize operators (functionalities) here, rather than wave types, because the wave types are not necessarily inherited in the operator itself, but in its implementation method on a particular architecture. As we will see, the same operator is implemented with

different spatial wave dynamic patterns on different architectures. The most important 2D operators, including all the CNN operators [1] are considered here. Description of these operators can be found in Appendix.

The first distinguishing feature is the location of active pixels [1]. If the active pixels are located along one or few one-dimensional stationary or propagating curves at a time, we call the operator front-active. If the active pixels are everywhere in the array, we call it area-active.

The common property of the front-active propagations is that the active pixels are located only at the propagating wave fronts [37]. This means that at the beginning of the wave dynamics (transient) some pixels become active, others remain passive. The initially active pixels may initialize wave fronts which start propagating. A propagating wave front can activate some further passive pixels. This is the mechanism how the wave proceeds. However, pixels apart from a waveform cannot become active [1], as we have seen it in Section 2.3. This theoretically enables us to compute only the pixels which are along the front lines, and do not waste efforts to the unchanging others. The question is which are the architectures that can take advantage of such a spatially selective computation.

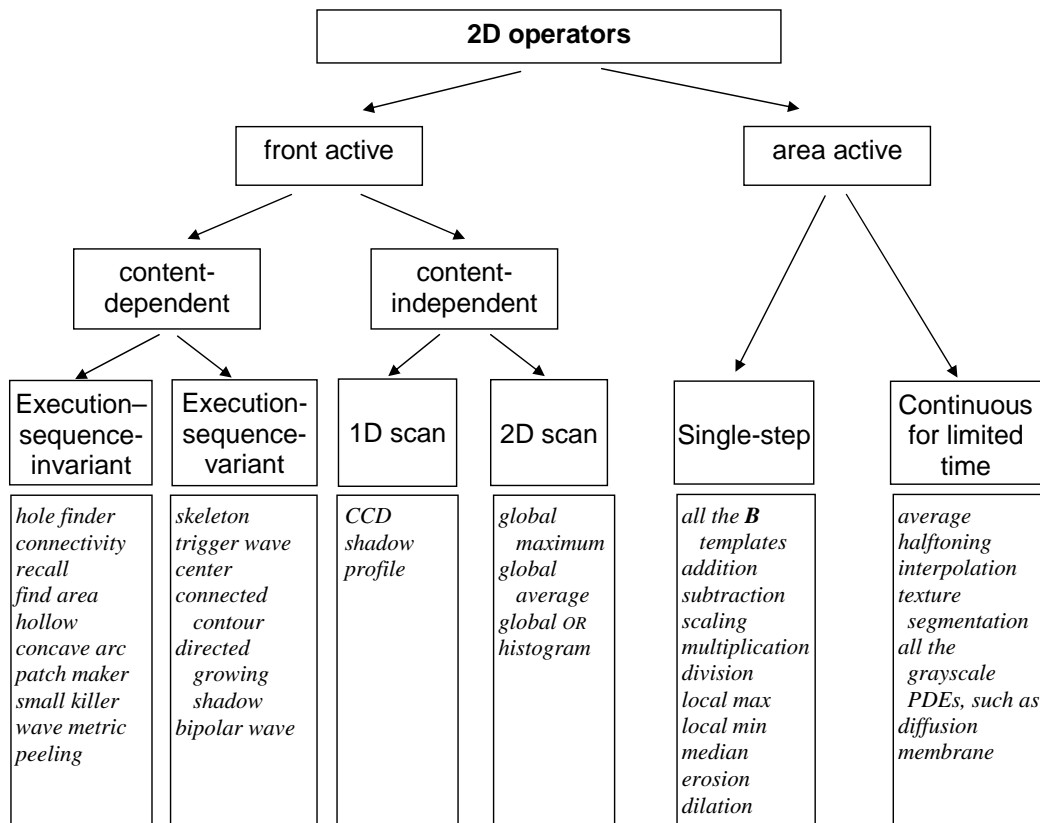


Figure 47. 2D local operator categorization

The front active operators such as *reconstruction*, *hole finder*, or *shadow* are typically binary waves. In CNN terms, they have binary inputs and outputs, positive self-feedback, and space invariant template values. Figure 47 contains three exemptions: *global max*, *global*

average, and *global OR*. These functions are not wave type operators by nature; however, we will associate a wave with them which solves them efficiently.

The front active propagations can be content-dependent or content-independent. The content-dependent operator class contains most of the operators where the direction of the propagation depends on the local morphological properties of the objects (e.g., shape, number, distance, size, connectivity) in the image (e.g., *reconstruct*). An operator of this class can be further distinguished as execution-sequence-variant (*skeleton*, etc) or execution-sequence-invariant (*hole finder*, *recall*, *connectivity*, etc). In the first case the final result may depend on the spatial-temporal dynamics of the wave, while in the latter it does not. Since the content-dependent operator class contains the most interesting operators with the most exciting dynamics, they are further investigated in Section 4.2.1.1.

We call the operators content-independent when the direction of the propagation and the execution time do not depend on the shape of the objects (e.g., *shadow*). According to propagation, these operators can be either one- (e.g., *CCD*, *shadow*, *profile* [23]) or two-dimensional (*global maximum*, *global OR*, *global average*, *histogram*). Content-independent operators are also called single-scan, for their execution requires a single scanning of the entire image. Their common feature is that they reduce the dimension of the input 2D matrices to vectors (*CCD*, *shadow*, *profile*, *histogram*) or scalars (*global maximum*, *global average*, *global OR*). It is worth to mention that on the coarse- and fine-grain topographic array processors the *shadow*, *profile* and *CCD* are content-dependent operators, and the number of the iterations (or analog transient time) depends on the image content only. The operation is completed, when the output is ceased to change. Generally, however, it is less efficient to include a test to detect a stabilized output, than to let the operator run in as many cycles as it would run in the worst case.

The area active operator category contains the operators where all the pixels are to be updated continuously (or in each iteration). A typical example is *heat diffusion*. Some of these operators can be solved in a single update of all the pixels (e.g., all the CNN **B** templates [23]), while others need a limited number of updates (*halftoning*, *constrained heat diffusion*, etc.).

The fine-grain architectures do update in every pixel location in fully parallel in each time instance. Therefore, the area active operators are naturally the best fit for these computing architectures.

4.2.1.1 Execution-sequence-variant versus execution-sequence-invariant operators

The crucial difference in fine-grain and pipe-line architectures is in their state overwriting methods. In the fine-grain architecture the new states of all the pixels are calculated in parallel, and then the previous one is overwritten again in parallel, before the next update

cycle is commenced. In the pipe-line architecture, however, the new state is calculated pixel-wise, and it is selectable whether to overwrite a pixel state before the next pixel is calculated (pixel overwriting), or to wait until the new state value is calculated for all the pixels in the frame (frame overwriting). In this context, update means the calculation of the new state for an entire frame. Figure 48 and Figure 49 illustrate the difference between the two overwriting schemes. In case of an execution-sequence-variant operation, the result depends on the frame overwriting schemes.

Here the calculation is done pixel-wise, left to right and row-wise top to down. As we can see, overwriting each pixel before the next pixel's state is calculated (pixel overwriting) speeds up the propagation in the directions of the proceeding of calculation.

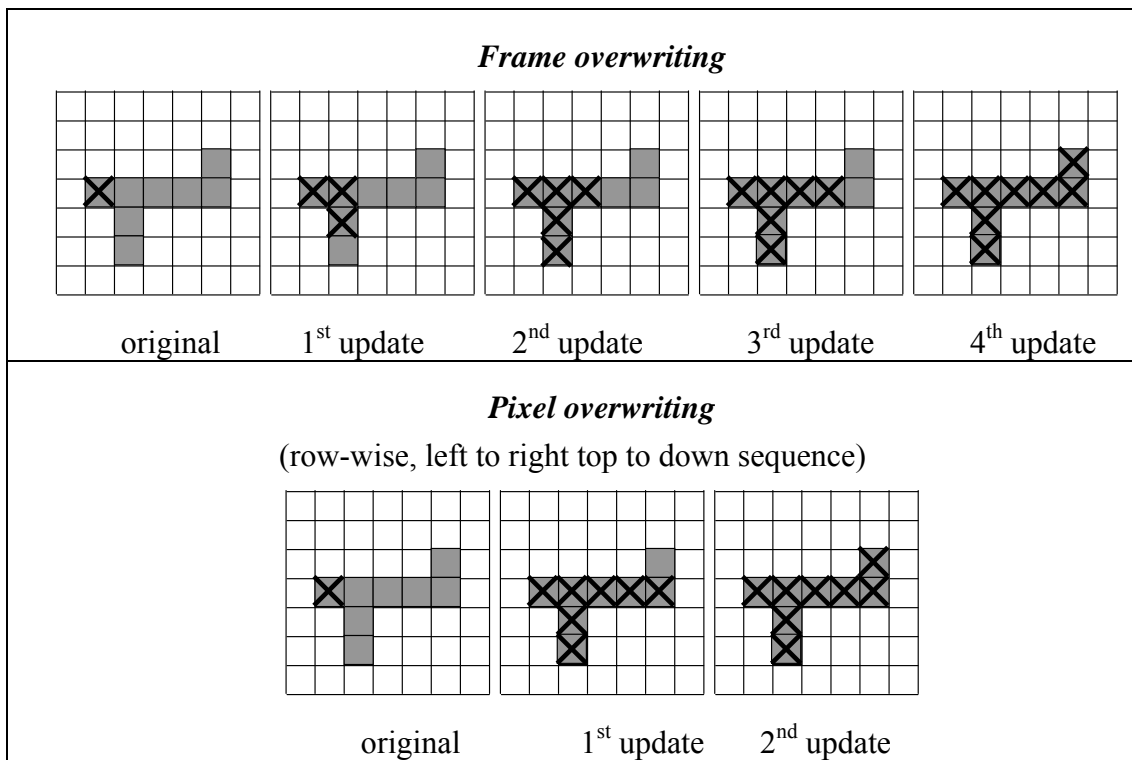


Figure 48. Execution-invariant sequence in different overwriting schemes. Given an image with grey objects against white background. The propagation rule is that the propagation starts from the marked pixel (denoted by X), and it can go on within the grey domain, proceeding one pixel in each update. In the figure, we can see the results of each update. Update means calculating the new states of all the pixels in the frame.

Based on the above, it is easy to draw the conclusion that the two updating schemes lead to two completely different propagation dynamics and final results in execution-variant cases. One is slower, but controlled, the other one is faster, but uncontrolled. The first can be used in cases when speed maximization is the only criterion, while the second is needed when the shape and the dynamics of the propagating wave front count. We called the former case

execution-sequence-invariant operators, the latter one execution-sequence-variant operators (Figure 47).

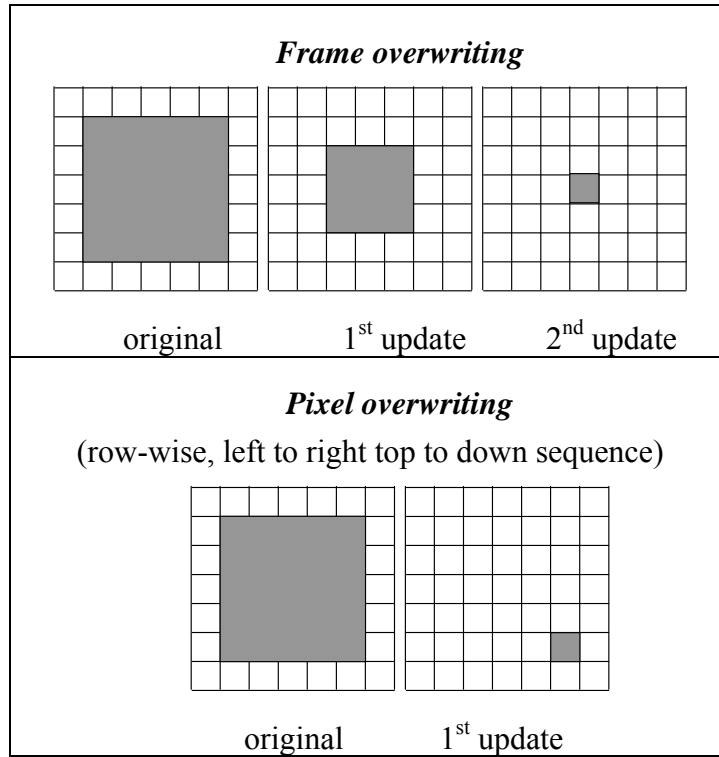


Figure 49. Execution-variant sequence in different overwriting schemes. Given an image with grey objects against white background. The propagation rule is that those pixels of the object, which has both object and background neighbor should become background. In this case, the subsequent peeling leads to find the centroid in the frame overwriting method, while it extracts one pixel of the object in the pixel overwriting mode.

In the fine-grain architecture we can use frame overwriting scheme only. In the coarse-grain architecture both pixel overwriting and frame overwriting methods can be selected within the individual sub-arrays. In this architecture, we may determine even the calculation sequence, which enables speed-ups in different directions in different updates. Later, we will see an example to illustrate how the *hole finder* operation propagates in this architecture. In the pipe-line architecture, we may decide which one to use, however, we cannot change the direction of the propagation of the calculation, unless paying significant penalty for it in memory size and latency time.

4.2.2 Processor utilization efficiency of the various operation classes

In this subsection, we will analyze the implementation efficiency of various 2D operators from different aspects. We will study both the execution methods and the efficiency from the processor utilization aspect. Efficiency is a key question, because in many cases one or a few wave fronts sweep through the image, and one can find active pixels only in the wave fronts,

which is less than one percent of the pixels, hence, there is nothing to calculate in the rest of image. We define a measure of efficiency of processor utilization with the following form:

$$\eta = O_r / O_t \quad (4.1)$$

where:

- O_r : the minimum number of required elementary steps to complete an operation, assuming that the inactive pixel locations are not updated
- O_t : is the total number of elementary steps performed during the calculation by all the processors in the particular processor architecture.

The efficiency of processor utilization figure will be calculated in the following where it applies, because this is a good parameter (among others) to compare the different architectures.

4.2.2.1 *Execution-sequence-invariant content-dependent front-active operators*

A special feature of content-dependent operators is that the path and length of the path of the propagating wave front drastically depend on the image contents itself. For example, the range of the necessary frame overwritings with a hole finder operation is from zero overwriting to $n/2$ in a fine-grain architecture, assuming $n \times n$ pixel array size. Hence, neither the propagation time, nor the efficiency can be calculated without knowing the actual image.

Since the gap between the worst and best case is extremely high, it is not meaningful to provide these limits. Rather, it makes more sense to provide approximations for certain image types. But before that, we examine how to implement these operators on the studied architectures. For this purpose, we will use the *hole finder* operator, as an example. Here we will clearly see how the wave propagation follows different paths, as a consequence of varying propagation speed corresponding to different directions. Since this is an execution-sequence-invariant operation, it is certain that wave fronts with different trajectories lead to the same good result.

The *hole finder* operation, that we will study here, is a “grass fire” operation, in which the fire starts from all the boundaries at the beginning of the calculation, and the boundaries of the objects behave like firewalls. In this way, at the end of the operation, only the holes inside objects remain unfilled.

The *hole finder* operation may propagate to any direction. On a **fine-grain architecture** the wave fronts propagate one pixel steps in each update. Since the wave fronts start from all the edges, they meet in the middle of the image in typically $n/2$ updates, unless there are large structured objects with long bays which may fold the grass fire into long paths. In case of a text for example, where there are relatively small non-overlapping objects (with diameter k) with large but not spiral like holes, the wave stops after $n/2 + k$ operations. In case of an

arbitrary camera image with an outdoor scene, in most cases $3*n$ updates are enough to complete the operation, because the image may easily contain large objects blocking the straight paths of the wave front.

On a **pipe-line architecture**, thanks to the pixel overwrite scheme, the first update fills up most of the background (Figure 50). Filling in the remaining background requires typically k updates, assuming the largest concavity size with k pixels. This means that on a pipe-line architecture, roughly $k+1$ steps are enough, considering small, non-overlapping objects with size k .

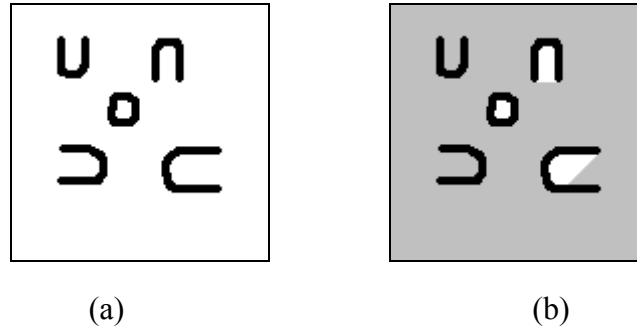


Figure 50. *Hole finder operation calculated with a pipe-line architecture. (a): original image. (b): result of the first update. (The freshly filled up areas are indicated with grey, just to make it more comprehensible. However, they are black on the black-and-white image, same as the objects.)*

In the **coarse-grain architecture** we can also apply the pixel overwriting scheme within the $N \times N$ sub-arrays (Figure 51). Therefore, within the sub-array, the wave front can propagate in the same way, as in the pipe-line architecture. However, it cannot propagate beyond the boundary of the sub-array, in a single update. In this way, the wave front can propagate N positions in the direction which correspond to the calculation directions, and one pixel in the other directions, in each update. In this way, in n/N updates, the wave-front can propagate n positions in the supported directions. However, the k sized concavities in other directions would require k more steps. To avoid these extra steps, without compromising the speed of the wave-front, we can switch between the top-down and the bottom-up calculation directions after each update. The resulting wave-front dynamics is shown in Figure 52. This means that for an image, containing only few, non-overlapping small objects with concavities, we need about $n/N+k$ steps to complete the operation.

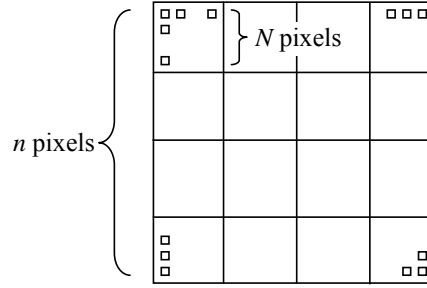


Figure 51. Coarse-grain architecture with $n \times n$ pixels. Each cell is to process an $N \times N$ pixel sub-array.

The **DSP-memory architecture** offers several choices depending on the internal structure of image. The simplest is to apply pixel overwriting scheme, and switch the direction of the calculation. In case of binary image representation, only the vertical directions (up or down) can be efficiently selected, due to the packed 32 pixel line segment storage and handling. In this way the clean vertical segments (columns of background with maximum one object) are filled up after the second update, and filling up the horizontal concavities would require k steps.

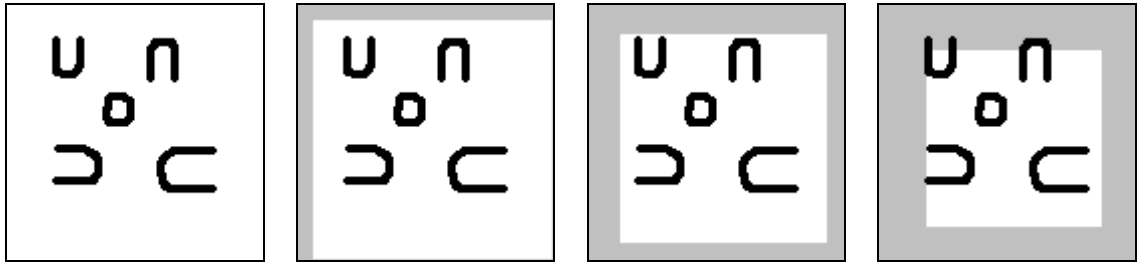


Figure 52. Hole finder operation calculated in a coarse-grain architecture. The first picture shows the original image. The rest shows the sequence of updates, one after the other. The freshly filled-up areas are indicated with grey (instead of black) to make it easier to follow the dynamics of calculation.

4.2.2.2 Execution-sequence-variant content-dependent front active operators

The calculation method of the execution-sequence-variant content-dependent front active operators is very similar to that of their execution-sequence-invariant counterparts. The only difference is that in each of the architectures the frame overwriting scheme should be used. This does not make any difference in fine-grain architectures, however, it slows down all the other architectures significantly. In the DSP-memory architectures, it might even make sense to switch to one byte/pixel mode, and calculate updates in the wave fronts only.

4.2.2.3 1D content-independent front active operators (1D scan)

In the 1D content-independent front active category, we use the vertical shadow (north to south) operation as an example. In this category, varying the orientation of propagation may cause drastic efficiency differences on the non-topographic architectures.

On a **fine-grain discrete time** architecture the operator is implemented in a way that in each time instance, each processor should check the value of its upper neighbor. If it is +1 (black), it should change its state to +1 (black), otherwise the state should not change. This can be implemented in one single step in a way, that each cell executes an *OR* operation with its upper neighbor, and overwrites its state with the result. This means that in each time instance the processor array executed n^2 operations, assuming $n \times n$ pixel array size.

In discrete time architectures, each time instance can be considered as a single iteration. In each iteration the shadow wave front moves by one pixel to the south, that is we need n steps for the wave front to propagate from the top row to the bottom (assuming boundary condition above the top row). In this way, the total number of operations, executed during the calculation is n^3 . However, the strictly required number of operations is n^2 , because it is enough to do these calculations at the wave front, only ones in each row, starting from the top row, and going down row by row, rolling over the results from the front line to the next one. In this way, the efficiency of the processor utilization in *vertical shadow* calculation in the case of fine-grain discrete time architectures is

$$\eta = 1/n \quad (4.2)$$

Considering computational efficiency, the situation is the same in **fine-grain continuous architectures**. However, from the point of power efficiency the Asynchronous Cellular Logic Network [47] is very advantageous, because only the active cells in the wave front consume switching power. Moreover, the extraordinary propagation speed (500 ps/cell) compensates for the low processor utilization efficiency.

If we consider a **coarse-grain architecture** (Figure 51), the *vertical shadow* operation is executed in a way that each cell executes the above *OR* operation from its top row, and goes on from the top downwards in each column. This means that $N \times N$ operations are required for a cell to process its sub-array. It does not mean, however, that in the first $N \times N$ steps the whole array is processed correctly, because only the first cell row has all the information for locally finalizing the process. For the rest of the rows their upper boundary condition have not “arrived”, hence at these locations correct operations cannot be performed. Thus, in the first $N \times N$ steps, the first N rows were completed only. However, the total number of operation executed by the array during this time is

$$O_{N \times N} = N * N * n/N * n/N = n * n, \quad (4.3)$$

because there are $n/N * n/N$ processors in the array, and each processor is running all the time. To process also the rest of the lines we need to perform

$$O_t = O_{N \times N} * n/N = n^3/N. \quad (4.4)$$

The resulting efficiency is:

$$\eta = N/n \quad (4.5)$$

It is worth to stop at this result for a while. If we consider a fine-grain architecture ($N=1$), the result is the same as we obtained in (4.2). Its optimum is $N=n$ (one processor per column) when the efficiency is 100%. It turns out that in case of *vertical shadow* processing, the efficiency increases by increasing the number of the processor columns, because in that case, one processor has to deal with less columns. However, the efficiency does not increase when the number of the processor rows is increased. (Indeed, one processor/column is the optimal, as it was shown.) Thought the unused processor cells can be switched off with minor extra effort to increase power efficiency, but it would certainly not increase processor utilization.

Pipe-line architecture as well as **DSP-memory architecture** can execute *vertical shadow* operation with 100% processor utilization, because there are no multiple processors in a column working parallel.

We have to note, however, that shadows to other three directions are not as simple as the one to downwards. In **DSP architectures**, *horizontal shadows* cause difficulties, because the operation is executed parallel on a $32 \times I$ line segment, hence only one of the positions (where the actual wave front is located) performs effectual calculation. If we consider a left to right shadow, this means that once in each line (at left-most black pixel), the shadow propagation should be calculated precisely for each of the 32 positions. Once the “shadow head” (the 32 bit word, which contains the left-most black pixel) is found, and the shadow is calculated within this word, the task is easier, because all the rest of the words in the line should be filled with black pixels, independently of their original content. Thus the overall resulting cost of a *horizontal shadow* calculation on a **DSP-memory architecture** can be even 20 times higher than that of a *vertical shadow* for a 128×128 sized image. Similar situation might happen in **coarse-grain architectures**, if they handle $n \times I$ binary segments.

While **pipe-line architectures** can execute the *left to right and top to bottom shadows* in a single update at each pixel location, the other directions would require n updates, unless the direction of the pixel flow is changed. The reason of such a high inefficiency is that in each update, the wave front can propagate only one step in the opposite direction.

4.2.2.4 2D content-independent front active operators (2D scan)

The operators belonging to the 2D content-independent front active category require simple scanning of the frame. In *global max* operation for example, the actual maximum

value should be passed from one pixel to another one. After we scanned all the pixels, the last pixel carries the global maximum pixel value.

In **fine-grain architectures** this can be done in two phases. First, in n comparison steps, each pixel takes over the value of its upper neighbor, if it is larger than its own value. After n steps, each pixel in the bottom row contains the largest value of its column. Then, in the second phase after the next n horizontal comparison steps, the global maximum appears at the end of the bottom row. Thus, to obtain the final result requires $2n$ steps. However, as a fine-grain architecture executes $n \times n$ operations in each step, the total number of the executed operations are $2n^3$. However, the minimum number of requested operation to find the largest value is n^2 only. Therefore, the efficiency in this case is:

$$\eta = 1/2n \quad (4.6)$$

The most frequently used operation in this category is *global OR*. To speed up this operation in the fine-grain arrays, a *global OR* net is implemented usually [42][27]. This $n \times n$ input OR gate requires minimal silicon space, and enables to calculate *global OR* in a single step (few microsecond).

However, in that case, when a fine-grain architecture is equipped with *global OR*, the global maximum can be calculated as a sequence of iterated *threshold* and *global OR* operations with interval halving (successive approximation) method applied parallel to the whole array. This means that a global threshold is applied first for the whole image at level $1/2$, and if there are pixels, which are larger than this, we will do the next global thresholding at $3/4$, and so on. Assuming 8 bit accuracy, this means that in 8 iterations (16 operations), the global maximum can be found. The efficiency is much better in this case:

$$\eta = 1/16$$

In **coarse-grain architectures**, each cell calculates the *global maximum* in its sub-array in $N \times N$ steps. Then n/N vertical steps come, and finally, n/N horizontal steps to find the largest values in the entire array. The total number of steps in this case is $N^2 + 2n/N$, and in each step, $(n/N)^2$ operations are executed. The efficiency is:

$$\eta = n^2 / (N^2 + 2n/N) * (n/N)^2 = 1 / (1 + 2n/N^3) \quad (4.7)$$

Since the sequence of the execution does not matter in this category, it can be solved with 100% efficiency in **pipe-line** and the **DSP-memory architectures**.

4.2.2.5 Area active operators

The area active operators require some computation in each pixel in each update; hence, all the architectures work with 100% efficiency. Since the computational load is very high here, it is the most advantageous for the many-core architectures, because the speed advantage of the many processor can be efficiently utilized.

4.2.3 Multi-scale processing

Generally, multi-scale processing technique is applied in those situations, when the calculation of an operator on a downscaled image leads to acceptable result from accuracy point of view. Since the calculation of the operator requires significantly smaller computational effort in a lower resolution, in many cases the downscaling, the upscaling (if needed), and the calculation on the downscaled domain requires less computational effort than the calculation of the operator in the original scale. Diffusion is a typical example for this.

Here we discuss how the approximation of the *diffusion* operator leads to a multi-scale representation, and analyze its implementation on the discussed architectures. However, with a similar approach, other binary or grayscale front- and area- active operators can be scaled down and executed, as well.

Two ways are used generally to compute the *diffusion* operator on topographic array computers. The first is the iterative way. The second way is to implement it on a hardwired resistive grid, as we have seen in analog fine-grain topographic architectures. Here we deal with the first option.

The problem with the iterative implementation of *diffusion* equation is that after a few iterations the differences of the neighboring pixels become very small, and the propagation slows down. Moreover, if there are some computational errors, due to the limited precision of the processors, calculation of the *diffusion* equation will be useless and irrelevant, after a while. To obtain accurate solution would require floating point number representation and a large number of iterations. However, one can approximate it by using multi-scale approach, as it is shown in Figure 53. As we can see, 10 iterations on a full scale image result in small blurring only, while the same 10 iterations on a downscaled image lead to large scale *diffusion*. The downscaling and the up scaling with linear interpolation need less computational effort, than a single iteration of the diffusion. Moreover, the calculation of an iteration on the downscaled image requires only $1/s^2$ (s is the downscaling factor) of computational power. Naturally, it should be kept in mind that this method can be used in that cases only when the accuracy of the approximated diffusion operator is good satisfactory in a particular application.

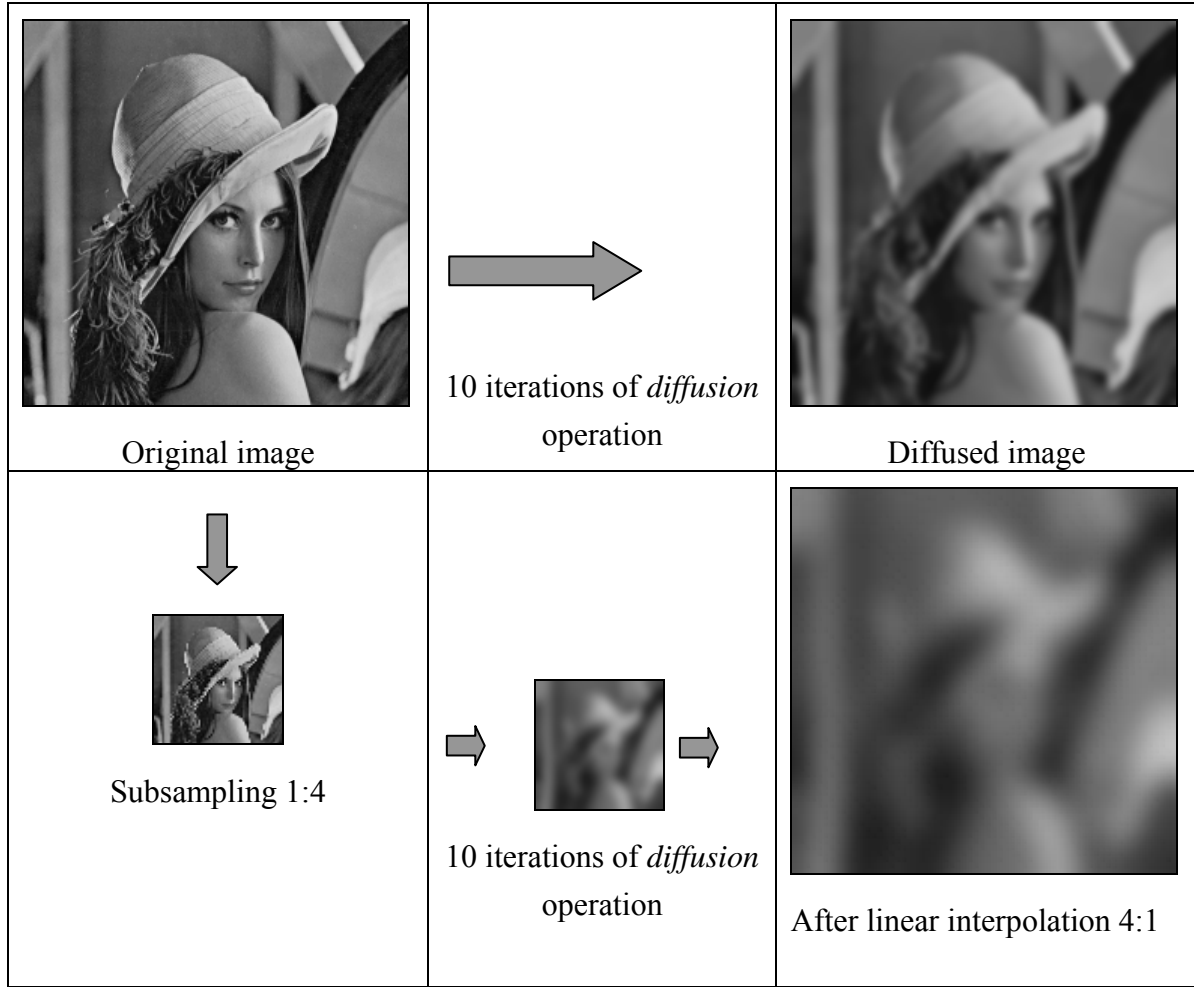


Figure 53. Iterative approximation of the diffusion operator combining different spatial resolutions.

The multi-scale iterative *diffusion* can be implemented on classic DSP-memory architectures, multi-core pipe-line architectures (Figure 54), and on coarse-grain architectures as well. In fine-grain architectures the multi-scale approach cannot be efficiently implemented.

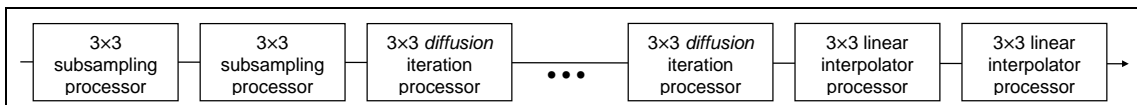


Figure 54. Implementation of multi-scale diffusion calculation approach on a pipe-line architecture. In this example, it starts with two subsampling steps. The pixel clock drops into $1/16^{th}$. Then the computationally hard diffusion calculations can be applied much easier since more time is available for each pixel. The processing is completed with the 2 interpolation steps.

4.3 Comparison of the architectures

As we have stated in the previous section, front active wave operators run well under 100% efficiency on topographic architectures, since only the wave fronts need calculation,

and the processors of the array in non-wave front positions do dummy cycles only or may be switched off. On the other hand, the computational capability (GOPs) and the power efficiency (GOPs/W) of multi-core arrays are significantly higher than those of DSP-memory architectures. In this section, we show the efficiency figures of these architectures in different categories. To make fair comparison with relevant industrial devices we have selected two market-leading, video processing units, a DaVinci video processing DSP from Texas Instruments (TMS320DM6443) [59], and a Spartan 3 DSP FPGA from Xilinx (XC3SD3400A) [64]. Both of these products' functionalities, capabilities and prices were optimized to efficiently perform embedded video analytics.

Table I summarizes the basic parameters of the different architectures, and indicates the processing time of a 3×3 convolution, and a 3×3 erosion. To make the comparison easier, values are calculated for images of 128×128 resolution. For this purpose, we considered 128×128 Xenon and Q-Eye chips. Some of these data are from catalogues, other ones are from measurements, or estimation. As fine-grain architecture examples, we included both the SCAMP and Q-Eye architectures.

We can see from Table I, the DSP was implemented on 90nm, while the FPGA on 65 nm technologies. In contrast Xenon, Q-Eye, and SCAMP were implemented on more conservative technologies and their power budget is an order of magnitude smaller. When we compare the computational power figures, we also have to take these parameters into consideration.

Table I shows the speed advantages of the different architectures, compared to DSP-memory architecture both in 3×3 neighborhood arithmetic (8 bit/pixel) and morphologic (1 bit/pixel) cases. This indicates the speed advantage of the area active single step, and the front active content-dependent execution-sequence-variant operators. In Table II, we summarize the speed relations of the rest of the wave type operations. The table indicates the computed values, using the formulas that we have derived in the previous section. In some cases, however, the coarse- and especially the fine-grain arrays contain some special accelerator circuits, which takes the advantage of the topographic arrangement and the data representation (e.g., global OR network, mean network, diffusion network). These are marked by notes, and the real speed-up with the special hardware is shown in parenthesis.

Table I Computational parameters of the different architectures for arithmetic (3×3 convolution) and logic (3×3 binary erosion) operations.

	DSP (DaVinci⁺)	Pipe-line (FPGA⁺⁺)	Coarse- grain (Xenon)	Fine-grain (SCAMP/Q- Eye)
<i>Silicon technology</i>	90nm	65nm	180nm	350/180nm
<i>Silicon area mm²</i>			100	100/50
<i>Power consumption</i>	1.25 W	2-3W	0.08 W	0.20 W
<i>Arithmetic proc. clock speed</i>	600 MHz	250 MHz	100 MHz	1,2 / 2.5 MHz
<i>Number of arithmetic proc.</i>	8	120	256	16384
<i>Efficiency of arithmetic calc.</i>	75% *	100%	80% ***	50% **
<i>Arit. computational speed</i>	3.6 GMAC	30 GMAC	20 GMAC	~20GOPS****
<i>3×3 convolution time</i>	42.3 μs*****	4.9 μs	12.1 μs	22 μs ****
<i>Arithmetic speed-up</i>	1	8.6	3.5	1.9
<i>Morph. proc. clock speed</i>	600 MHz	83 MHz	100 MHz	1,2 / 5 MHz
<i>Number of morphologic proc.</i>	64	864	2048	147456
<i>Morphologic processor kernel type</i>	2 × 32 bit	96 × 9 bit	256 × 8 bit	16384 × 9 bit
<i>Efficiency of morphological calc.</i>	28% *	100%	90% ***	100%
<i>Morphologic computational power</i>	10 GOPS	71 GOPS	184 GOPS	737 GOPS
<i>3×3 morphologic operation time</i>	13.6 μs*****	2.05 μs	1.1 μs	0.2 μs
<i>Morphologic speed-up</i>	1	6.6	12.4	68.0

⁺ Texas Instrument DaVinci video processor (TMS320DM64x)

⁺⁺ Xilinx Spartan 3ADSP FPGA (XC3SD3400A)

* processors are faster than cache access

** data access from neighboring cell is an additional clock cycle

*** due to pipe-line stages in the processor kernel, (no effective calculation in each clock cycle)

**** no multiplication, scaling with few discrete values

***** these data-intensive operators slow down to 1/3rd or even 1/5th when the image does not fit to the internal memory (typically above 128×128 with a DaVinci, which has 64kByte internal memory)

In our comparison tables, we have represented a typical FPGA as a vehicle to implement the pipe-line architectures. The only reason is that all the currently available pipe-line architectures are implemented in FPGAs is mainly attributed to much lower costs and quicker time-to-market development cycles. However, they could also be certainly implemented in ASIC, which would significantly reduce their power consumption, and decrease their large-volume prices making it possible to process even multi-mega pixel images at a video rate.

Table II Speed relations in the different function groups calculated for 128×128 sized images. The notes indicate the functionalities by which the topographic arrays are speeded up with special purpose devices.

	DSP (DaVinci ⁺)	Pipe-line (FPGA ⁺⁺)	Coarse-grain (Xenon)	Fine-grain discrete time (SCAMP/ Q-Eye)	Fine-grain continuous time (ACLA)
<i>1D content-independent front active operators</i>					
processor util. efficiency	100%	100%	N/n: 6.25%	1/n: 0.8%	1/n: 0.8%
speed-up in advantageous direction (vertical)	1	6.6	0.77	0.53	188
speed-up in disadvantageous direction (horizontal)	1	1	2	10.6	3750
<i>2D content-independent front active operators</i>					
processor util. efficiency	100%	100%	$1/(1+2n/N^3)$: 66%	1/2n: 0.4%	-
speed-up (<i>global OR</i>)	1	6.6	8.2 (13*)	0.27 (20*)	n/a
speed-up (<i>global max</i>)	1	8.6	2.3	n/a	n/a
speed-up (<i>average</i>)	1	8.6	2.3	n/a (2.5)**	n/a
<i>Execution-sequence- invariant content- dependent front active operators</i>					
<i>hole finder</i> with k=10 sized small objects	4 updates	k+1 updates (11)	n/N+k (26)	n/2+k updates (74)	n/2+k updates (74)
speedup	1	2.4	1.9	3.7	1500
<i>Area active</i>					
processor util. efficiency	100%	100%	100%	100%	
speedup	1	8.6	3.5	1.9 (210***)	n/a
<i>Multi-scale</i>					
1:4 scaling	1	8.6	3.5	0.1	n/a

⁺ Texas Instrument DaVinci video processor (TMS320DM64x)

⁺⁺ Xilinx Spartan 3ADSP FPGA (XC3SD3400A)

* Hard wired global OR device speeds up this function (<1 μs concerning the whole array)

** Hard wired mean calculator device makes this function available (~2 μs concerning the whole array)

*** Diffusion calculated on resistive network (<2 μs concerning the whole array)

Table III shows the computational power, the consumed power and the power efficiency of the selected architectures. As we can see, the three topographic arrays have over hundred times power efficiency advantage compared to DSP-memory architectures. This can be

explained with their local data access, and relatively low clock frequency. In case of ASIC implementation, the power efficiency of the pipe-line architecture would also be increased with a similar factor.

Table III Computational power, and the consumed electronic power, and their proportion in different architectures for convolution operations. As a comparison, the Cell Multiprocessor developed by IBM-Sony-Toshiba [57] is also given.

	GOPs	W	GOPs/W
DaVinci	3.6	1.25	2.88
Pipe-line (FPGA)	30	3	10
Xenon (64x64)	10	0.02	500
SCAMP (128x128)	20	0.2	100
Q-Eye	25	0.2	125
Cell multiprocessor	225	85	2.6

Figure 55 shows the relation between the frame-rate and the resolution in a video analysis task. Each of the processors had to calculate 20 *convolutions*, 2 *diffusions*, 3 *means*, 40 *morphologies* and 10 *global ORs*. Only the DSP-memory and pipe-line architectures support trading between resolution and frame-rate. The characteristics of these architectures form lines. The chart shows the performance of the three discussed chips too. The chips are represented here with their real sizes.

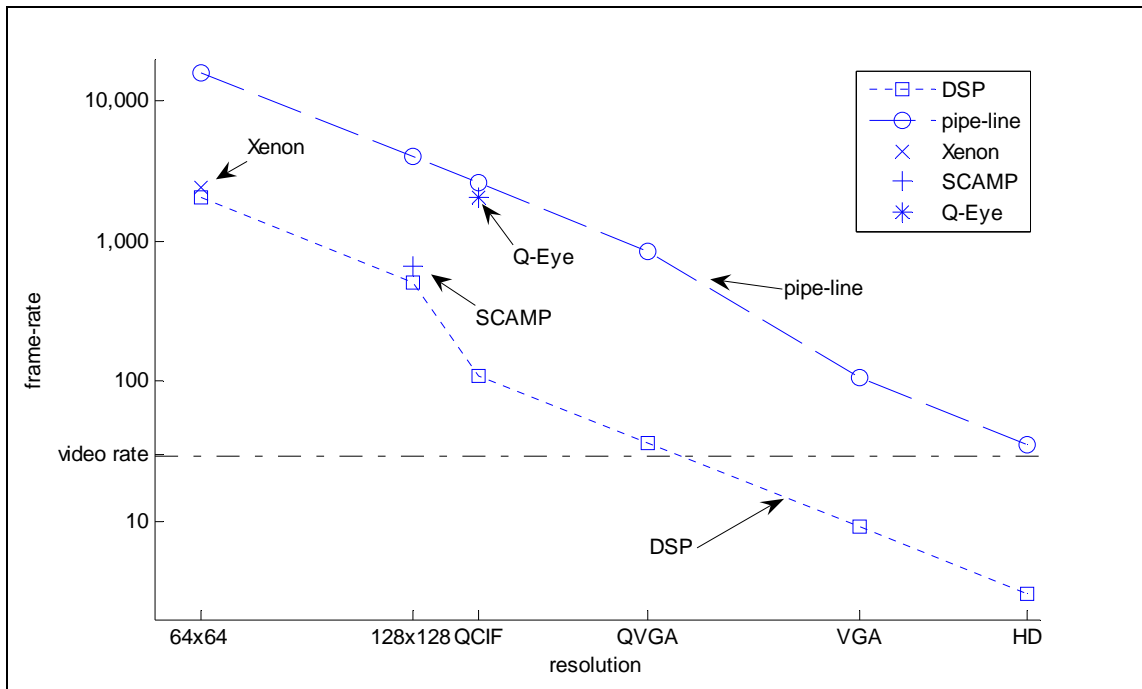


Figure 55. Frame-rate versus resolution in a typical image analysis task. Both of the axes are in logarithmic scale.

As it can be seen in Figure 55, both SCAMP and Xenon have the same speed as the DSP. In the case of Xenon, this is so, because its array size is 64×64 only. In the case of SCAMP, the processor was designed for very accurate low power calculation by using a conservative technology.

4.4 Optimal architecture selection

So far, we have studied how to implement the different wave type operators on different architectures, identified constraints and bottlenecks, and analyzed the efficiency of these implementations. After having these results in our hand, we can define rules for optimal image processing architecture selection for topographic problems.

Image processing devices are usually special purpose architectures, optimized for solving specific problems or a family of similar algorithms. Figure 56 shows a method of special purpose processor architecture selection. It always starts with the understanding of the problem in all aspect. Then, different algorithms suitable for solving the problem are derived. The algorithms are described with flowchart, with the list of the used operations, and with the specification of the most important parameters. In this way, a set of formal data describes the algorithms, which are as follows: **resolution**, **frame-rate**, **pixel clock**, **latency**, **computational demand** (type and number of operators), **and flowchart**. Other application-specific (secondary) parameters are also given: maximal **power consumption**, maximal **volume**, **economy** etc. The algorithm derivation is a human activity supported by various simulators for evaluation and verification purposes.

The next step is the architecture selection. By using the previously compiled data, we can define a methodology for the architecture selection step. As we will see, based on the formal specifications, we can derive the possible architectures. There might not be any, there might be exactly one, or there might be several, according to the demands of the specification of the algorithm.

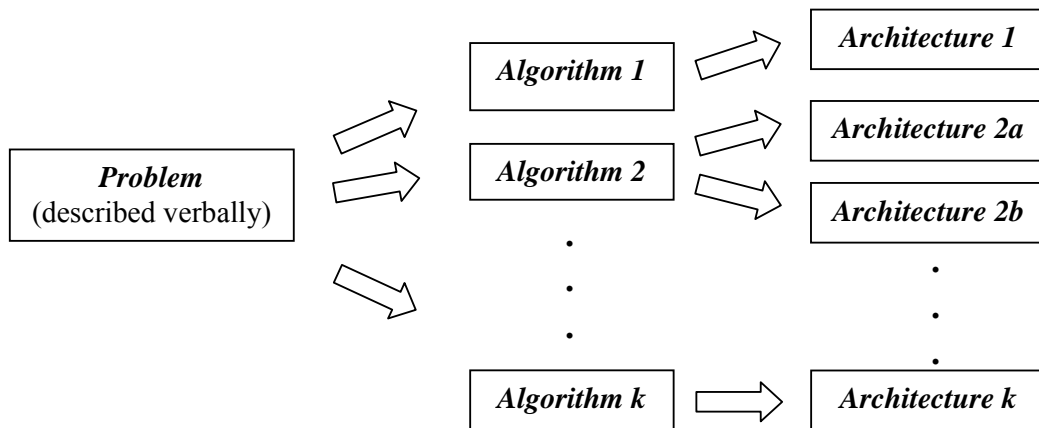


Figure 56. Methodology of special purpose processor architecture selection

The first step of the method is the comprehensive analysis of the parameter set. Fortunately, in many cases it immediately leads to a single possible architecture. If it does not lead to any architecture, in a second step, we have to seek for options, how to fulfill the original specification demands. If it leads to multiple architectures, a ranking is needed based on secondary parameters.

The three most important parameters are the *frame-rate*, the *resolution*, and their product, the minimal value of the *pixel clock*.² In many cases, especially in challenging applications, these parameters determine the available solutions. Figure 57 shows frame-rate – resolution matrix. The matrix is divided into 16 segments, and each segment indicates the potential architectures that can operate in that particular parameter environment. The matrix shows the minimal pixel clock figures (red) in the grid points also.

In Figure 57, the pipe-line and the DSP can be positioned freely between frame-rate and resolution without constrains. Thus they appear everywhere, under a certain pixel clock rate. The digital coarse-grain sensor-processor arrays appear in the low resolution part (left column), while the analog (mixed-signal) fine-grain sensor-processor arrays appear in both the low and medium resolution columns.

The next important parameter is the *latency*. Latency is critical when the vision device is in a control loop, because large delays might make the control loops instable. It is worth to distinguish three latency requirement regions:

- very low latency ($latency < 2ms$; e.g. missile, UAV, high speed robot controlling);
- low latency ($2ms < latency < 50ms$; e.g. robotic, automotive);
- high latency ($50ms < latency$; e.g. security, industrial quality check).

Latency has two components. The first is the readout time of the sensor, and the second is the completion of the processing on the entire frame. The readout time is negligible in the fine-grain mixed-signal architectures, since the analog sensor readout should be transferred to an analog memory through a fully parallel bus. The readout time is also very small ($\sim 100\mu s$) in the coarse-grain digital processor array, because there is an embedded AD converter array to do conversion in parallel. The DSPs and the pipe-line processor arrays use external image sensors, in which the readout time usually is in the millisecond range. Therefore, in case of very low latency requirements, the mixed-signal and the digital focal plane arrays can be used. (There are some ultra-high frame-rate sensors with high speed readout, which can be

² The minimal value of the pixel clock is equivalent to the product of the frame-rate and the number of pixels (resolution). If the image source is a sensor, the pixel clock of the processor is defined by the pixel clock of the sensor. Since there are short blank periods in the sensor readout protocol for synchronization purposes, the pixel clock is slightly higher than the minimal pixel clock even in those cases, when the integration is done parallel with the readout (CMOS or CCD rolling shutter mode). However, in low light applications, the integration time is much longer than the readout time. In these cases, the sensor pixel clock can be orders of magnitude higher than the minimal pixel clock.

combined with pipe-line processors. However, these can be applied in very special applications only due to their high complexities and costs.)

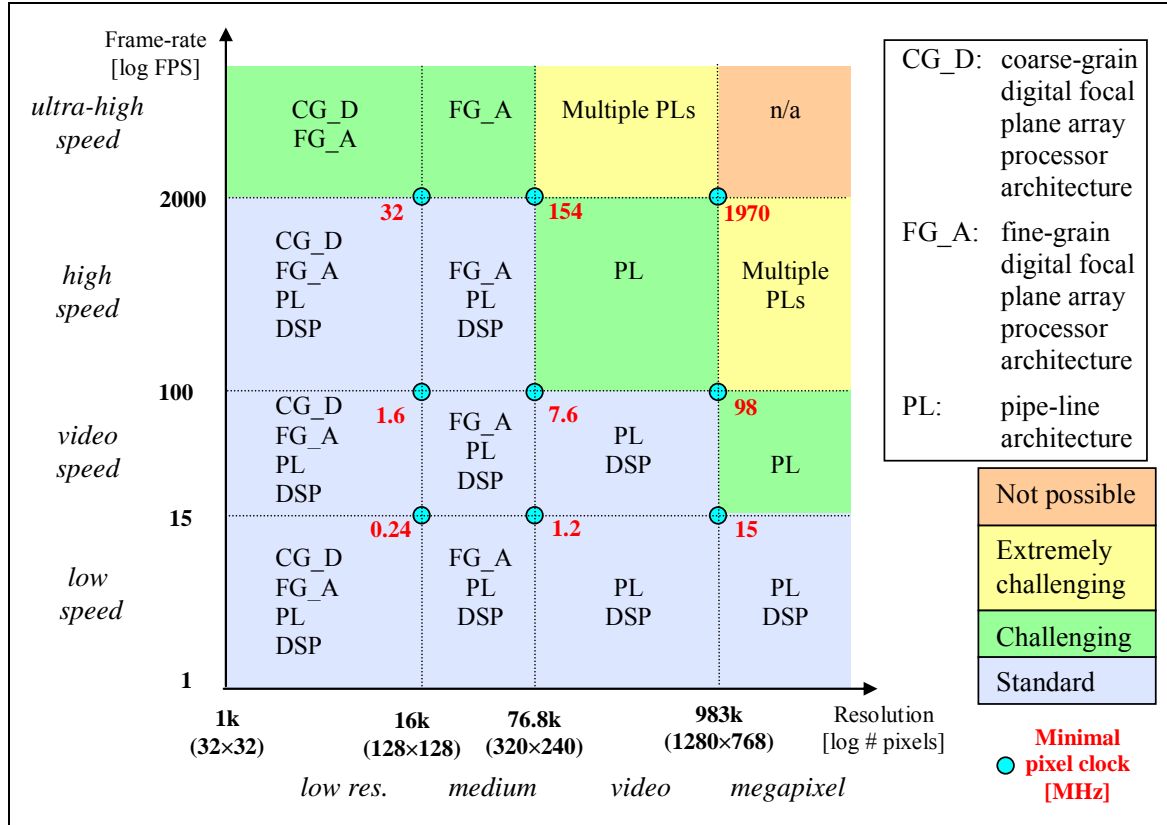


Figure 57. Feasible architectures in the frame-rate – resolution matrix

In the low latency category, those architectures can be used only, in which the sensor readout time plus the processing time is smaller than the latency requirements. In the high latency region, the latency does not mean any bottleneck.

The next descriptor of the algorithms is the **computational demand**. It is a list of the applied operations. Using the execution time figures that we calculated for different operations on the examined architectures, we can simply calculate the total execution time. (In case of the pipe-line architecture the delay of the individual stages should be summed up.) The total processing time should satisfy the following two relations:

$$t_{total_processing} < t_{latency} - t_{readout}$$

$$t_{total_processing} < 1/frame_rate$$

The last primary parameter is the **program flow**. The array processors and the DSP are not sensitive for branches in program flow. However, as we have already seen in Section 3.3.3, the pipe-line architectures are challenged by the program flow branches. As shown in Figure 34, implementation of a conditional branch needs the insertion of a frame buffer, which drastically increases the latency and makes the design more complex.

There are three secondary design parameters. The first is the **power consumption**. Generally, the ASIC solutions need much less power than the FPGA or DSP solutions. The second is the cubature of the circuit. Smaller cubature can be achieved with sensor-processor arrays, because the combination of these two functionalities reduces the chip count. The third parameter is the economy. In case of low volume, the DSP is the cheapest, because the invested engineering cost is the smaller there. In case of medium volume, the FPGA is the most economical, while in case of high volume, the ASIC solutions are the cheapest.

4.5 Summary

I have categorized the 2D operators into 6 sets, based on their implementation methods on different image processing architectures. By using this categorization, the efficiency figures of 2D operators were calculated considering different architectures. This enabled us to compare the architectures, and provide a guide for selecting the optimal architecture for a given algorithm. Moreover, we have measured, collected, or calculated some key parameters of existing implementations. Comparing the different architectures, we can draw the following conclusions:

- The computational speed on digital coarse-grain architectures is roughly the same as on fine-grain architectures (Figure 55). The accuracy of the digital one is better, however, the required silicon area is also larger (Table I).
- The analog/mixed signal fine-grain architecture can take advantage of utilizing various specific processing networks, like mean grid, diffusion grid, global OR grid, etc (Table II).
- In focal-plane sensor-processor application where the specification requires lower precision, the analog fine-grain implementations are more advantageous.
- In applications where high-precision calculation is required, the coarse-grain architecture is more advantageous.
- It is important to note that in the case of array processors, the speed up rate changes with the processor array size. In some cases the speed advantage is proportional to the number of the processors in the array (area active single step, and the front active content-dependent execution-sequence-variant operators), while in the rest of the cases, it is proportional with the number of the processors located in one row/column (Table II).
- As it is shown in Table III, the GOPs/W figure of the studied topographic many-core architectures are orders of magnitude better than the single or many core high-end processors used nowadays in PCs and servers,. This makes any of those much more suitable for embedded mobile applications, compared to a DSP or a RISC processor.

4.6 Conclusions

The classification of the 2D topographic operators and their efficiency calculation on different architectures is a new result. I foresee that this study will help researchers inside or outside the CNN community to deeply understand the connection between the 2D operators and the topographic architectures. They will learn what price they have to pay for selecting an exotic operator or applying an unusual branch in the flowchart. This knowledge will help them to optimally select architectures, or to avoid dead ends of projects due to an unfortunate architecture selection.

Acknowledgement

I owe thanks to Professor Tamás Roska for his kind help in my work in my entire academic carrier, for the warm and inspiriting atmosphere he permanently makes around himself, and for his constant but friendly pressure on me, to prepare this Dissertation.

I thank Dr András Radványi, who gave me a continuous help in the details of the preparation of the Dissertation.

I thank my local collages Péter Szolgay, Péter Földesy, Csaba Rekeczky, István Szatmári, Szabolcs Tőkés, and László Orzó, and my international collages professor Ángel Rodríguez-Vázquez, Gustavo Linnan, Ricardo Carmona, Piotr Dudek, Bertran Shi, Marco Gili, Paolo Arena, and Ari Paasio, for helping my research work in the last 10 years.

I thank my family, my wife Szilvia, and my sons, Álmos, Levente, and Botond, and my parents my Mom and my Dad who always supported my work and accepted the inconveniencies of my travels and late night or weekend works.

I greatly admire the supportive environment of my research institute, the Computer and Automation Research Institute of the Hungarian Academy of Sciences.

References

Publications of the Author

- [1] Á. Zarándy, "The Art of CNN Template Design", Int. J. Circuit Theory and Applications - Special Issue: Theory, Design and Applications of Cellular Neural Networks: Part II: Design and Applications, (CTA Special Issue - II), Vol.17, No.1, pp.5-24, 1999
- [2] Á. Zarándy, P. Keresztes, T. Roska, and P. Szolgay, "CASTLE: An emulated digital architecture; design issues, new results", Proceedings of 5th IEEE International Conference on Electronics, Circuits and Systems, (ICECS'98), Vol. 1, pp. 199-202, Lisboa, 1998
- [3] P. Keresztes, Á. Zarándy, T. Roska, P. Szolgay, T. Bezák, T. Hídvégi, P. Jónás, A. Katona, "An emulated digital CNN implementation", Journal of VLSI Signal Processing Special Issue: Spatiotemporal Signal Processing with Analogic CNN Visual Microprocessors, (JVSP Special Issue), Kluwer, 1999 November-December
- [4] Á. Zarándy, T. Roska: „Videojel feldolgozó számítógép, celluláris csip és eljárás egy vagy több bejövő videojelnek egy vagy több kimenő videojellé való átalakítására 2000
- [5] Á. Zarándy, T. Roska Video signal processing computer, cellular chip and method, 2002
- [6] P. Földesy, Á. Zarándy, Cs. Rekeczky, T. Roska: „Jelérzékelő és feldolgozó rendszer és eljárás 2005
- [7] P. Földesy, Zarándy, Cs. Rekeczky, T. Roska System and method for sensing and processing electromagnetic signals, 2006
- [8] Á. Zarándy, Cs. Rekeczky „Bi-i: a Standalone Cellular Vision System, Part I. Architecture and Ultra High Frame Rate Processing Examples” Proceedings of the CNNA-2004 Budapest, Hungary
- [9] Cs. Rekeczky, Á. Zarándy, „Bi-i: a Standalone Cellular Vision System, Part II. Topographic and Non-topographic Algorithms and Related Applications”, Proceedings of the CNNA-2004 Budapest, Hungary
- [10] Á. Zarándy, Cs. Rekeczky, P. Földesy, and I. Szatmári, "The New Framework of Applications - The Aladdin System", Journal of Circuits, Systems, and Computers (JCSC), Vol. 12, No. 6 (December 2003)
- [11] Á. Zarándy, R. Domínguez-Castro, and S. Espejo, "Ultra-high Frame Rate Focal Plane Image Sensor and Processor", IEEE Sensors Journal, Vol. 2, No. 6 pp.:559-565, December 2002

- [12] P. Földesy, Á. Zarándy, Cs. Rekeczky, and T. Roska, “Digital implementation of cellular sensor-computers”, *Int. J. Circuit Theory and Applications (CTA)*, Volume 34 , Issue 4, Pages: 409 – 428, July 2006
- [13] P. Földesy, Á. Zarándy, Cs. Rekeczky, and T. Roska „Configurable 3D integrated focal-plane sensor-processor array architecture”, *Int. J. Circuit Theory and Applications (CTA)*, pp: 573-588, 2008
- [14] P. Földesy, Á. Zarándy, Cs. Rekeczky and T. Roska “High performance processor array for image processing”, *ISCAS 2007*, New Orleans.
- [15] P. Földesy, Á. Zarándy, R. Carmona, Cs. Rekeczky, T. Roska, A. Rodriguez-Vazquez, “A 320x240 sensor-processor chip for air-born surveillance and navigation, submitted to *ECCTD 2009*
- [16] Á. Zarándy, Cs. Rekeczky, P. Földesy, „Analysis of 2D operators on topographic and non-topographic processor architectures”, *Proceedings of the CNNA-2008 Santiago de Compostella, Spain*
- [17] Á. Zarándy, Cs. Rekeczky „ Implementation and efficiency analysis of 2D operators on topographic and non-topographic image processor architectures”, *Int. J. Circuit Theory and Applications (CTA)*, paper submitted 2007
- [18] Á. Zarándy, Cs. Rekeczky, “Low-power processor array design strategy for solving computationally intensive 2D topographic problems” book edited by C. Baatar, W. Porod, and T. Roska, Springer (under publication)
- [19] A. Zarándy, Cs. Rekeczky: „Bi-i: A Standalone Ultra High Speed Cellular Vision System”, *IEEE Circuits and Systems Magazine*, second quarter 2005, pp36-45., 2005
- [20] Linan-Cembrano, G., Carranza, L., Rind, C, Zarandy, A., Soininen, M., Rodriguez-Vazquez, A, “Insect-Vision Inspired Collision Warning Vision Processor for Automotive”, *IEEE Circuits and Systems Magazine*, Volume: 8, Issue: 2 On page(s): 6-24 2008
- [21] L.O. Chua, T. Roska, T. Kozek, Á. Zarándy “CNN Universal Chips Crank up the Computing Power”, *IEEE Circuits and Devices*, July 1996, pp. 18-28, 1996.
- [22] Cs. Rekeczky, J. Mallett, Á. Zarándy, „Security Video Analytics on Xilinx Spartan -3A DSP”, *Xcell Journal*, Issue 66, fourth quarter 2008, pp: 28-32.
- [23] T. Roska, L. Kék, L. Nemes, Á. Zarándy, M. Brendel and P. Szolgay, "CNN Software Library (Templates and Algorithms) Version 7.2", (DNS-1-1998), Budapest, MTA SZTAKI, 1998, http://cnn-technology.itk.ppke.hu/Library_v2.1b.pdf

Other references

- [24] L.O. Chua and L. Yang, “Cellular Neural Networks: Theory and Applications”, IEEE Transactions on Circuits and Systems, vol. 35, no. 10, October 1988, pp. 1257-1290, 1988.
- [25] L. O. Chua, T. Roska, “Cellular Neural Networks and Visual Computing”, Cambridge University Press, 2002
- [26] L.O. Chua and T. Roska, “The CNN Paradigm”, IEEE Transactions on Circuits and Systems - I, vol. 40, no. 3, March 1993, pp. 147-156, 1993.
- [27] T. Roska and L.O. Chua, “The CNN Universal Machine: An Analogic Array Computer”, IEEE Transactions on Circuits and Systems - II, vol. 40, March 1993, pp. 163-173. 1993.
- [28] K.R.Crounse, L.O.Chua, "Efficient The CNN Universal Machine is as universal as a Turing Machine ", IEEE Trans. Circuits and Systems I. Volume 43, Issue 4, Apr 1996 Page(s):353 – 355
- [29] L. Nemes, L. O. Chua , T. Roska, “Implementation of arbitrary Boolean functions on a CNN Universal Machine”, Int. J. Circuit Theory and Applications, Special Issue: Theory, Design and Applications of Cellular Neural Networks, Volume 26, Issue 6 , Pages 593 – 610.
- [30] N. Takashashi, L.O. Chua, “On the complete stability of non-symmetric cellular neural networks”, Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on Circuits and Systems I Volume: 45, Issue: 7 On page(s): 754-758
- [31] T. Roska, “Circuits, computers, and beyond Boolean logic”, Int. J. Circuit Theory and Applications, Volume 35 Issue 5-6, Pages 485 – 496, 2007
- [32] L. Belady, T. Roska “Virtual Cellular Machines - the Virtual Processor Array concept with mega processor computers via kilo-core chips” in preparation
- [33] Kozek, T. Roska, T. Chua, L.O., “Genetic algorithm for CNN template learning”, IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, Jun 1993, Volume: 40, Issue: 6, On page(s): 392-402
- [34] Nossek, J.A., “Design and learning with cellular neural networks”, Cellular Neural Networks and their Applications. CNNA-94., page(s): 137-146, 1994
- [35] S. Espejo, R. Carmona, R. Domínguez-Castro and A. Rodríguez-Vázquez “A VLSI-Oriented Continuous-Time CNN Model”, International Journal of Circuit Theory and Applications, Vol. 24, pp. 341-356, May-June 1996.
- [36] S.Espejo, A.Rodríguez-Vázquez, R.Dominguez-Castro, and R.Carmona “Convergence and Stability of FSR CNN Model” Proc. of the IEEE Conference on Cellular Neural Networks and their Applications (CNNA-96), pp. 411-416. Seville, 1996.

- [37] Cs. Rekeczky and L. O. Chua, "Computing with Front Propagation: Active Contour and Skeleton Models in Continuous-time CNN", *Journal of VLSI Signal Processing Systems*, Vol. 23, No. 2/3, pp. 373-402, November-December 1999.
- [38] H. Aomori, T. Otake, N. Takahashi, M. Tanaka "Sigma-Delta Cellular Neural Network for 2-D Modulation" *IJCNN 2007*, Orlando, Florida, August 12-17,
- [39] J.M.Cruz, L.O.Chua, and T.Roska, "A Fast, Complex and Efficient Test Implementation of the CNN Universal Machine", *Proc. of the third IEEE Int. Workshop on Cellular Neural Networks and their Application (CNNA-94)*, pp. 61-66, Rome Dec. 1994.
- [40] H.Harrer, J.A.Nossek, T.Roska, L.O.Chua, "A Current-mode DTCNN Universal Chip", *Proc. of IEEE Intl. Symposium on Circuits and Systems*, pp.135-138, 1994.
- [41] A. Paasio, A. Dawindzuk, K. Halonen, V. Porra, "Minimum Size 0.5 Micron CMOS Programmable 48x48 CNN Test Chip" *European Conference on Circuit Theory and Design*, Budapest, pp. 154-15, 1997.
- [42] Gustavo Liñan Cembrano, Ángel Rodríguez-Vázquez, Servando Espejo-Meana, Rafael Domínguez-Castro: ACE16k: A 128x128 Focal Plane Analog Processor with Digital I/O. *Int. J. Neural Syst.* 13(6): 427-434 (2003)
- [43] S. Espejo, R. Carmona, R. Domínguez-Castro, and A. Rodríguez-Vázquez, "CNN Universal Chip in CMOS Technology", *Int. J. of Circuit Theory & Appl.*, Vol. 24, pp. 93-111, 1996
- [44] S. Espejo, R. Domínguez-Castro, G. Liñán, Á. Rodríguez-Vázquez, "A 64x64 CNN Universal Chip with Analog and Digital I/O", in *Proc. ICECS'98*, pp. 203-206, Lisbon 1998
- [45] R. Carmona, S. Espejo, R. Domínguez Castro, A. Rodríguez Vázquez, T. Roska, T. Kozek, L. O. Chua, "An 0.5- μ m CMOS analog random access memory chip for TeraOPS speed multimedia video processing", *IEEE Transactions on Multimedia*, Volume 1, Issue 2, Jun 1999 Page(s):121-135.
- [46] P.Dudek "An asynchronous cellular logic network for trigger-wave image processing on fine-grain massively parallel arrays", *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*,. 53 (5): pp. 354-358, 2006.
- [47] A. Lopich, P. Dudek, "Implementation of an Asynchronous Cellular Logic Network As a Co-Processor for a General-Purpose Massively Parallel Array", *ECCTD 2007*, Seville, Spain.
- [48] A. Lopich, P. Dudek., " Architecture of asynchronous cellular processor array for image skeletonization", *Circuit Theory and Design*, Volume: 3, On page(s): 81-84, 2005.

- [49] P.Dudek and S.J.Carey, "A General-Purpose 128x128 SIMD Processor Array with Integrated Image Sensor", Electronics Letters, vol.42, no.12, pp.678-679, June 2006
- [50] Z. Nagy, P. Szolgay "Configurable Multi-Layer CNN-UM Emulator on FPGA" IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, Vol. 50, pp. 774-778, 2003
- [51] T. Szirányi, M. Csapodi: "Texture Classification and Segmentation by Cellular Neural Network using Genetic Learning", (CVGIP) Computer Vision and Image Understanding, volume 71, No , pp255-270, September, 1998.
- [52] J. Fernández-Berni, R. Carmona-Galán, "Practical Limitations to the Implementation of Resistive Grid Filtering in Cellular Neural Networks", ECCTD 2007, Seville, Spain
- [53] P. P. Civalleri, M. Gilli, "Global dynamic behaviour of a three-cell connected component detector CNN", International Journal of Circuit Theory and Applications, Volume 23, Issue 2 , Pages 117 – 135
- [54] M. Minsky, S. Papert, "Perceptrons: An Introduction to Computational Geometry", MIT Press, Cambridge, MA, 1969.
- [55] E.R. Kandel, J.H. Schwartz, "Principles of Neural Science", second edition, Elsevier, New York, Amsterdam, Oxford, 1985
- [56] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. D. Kubiatowicz, E. A. Lee, N. Morgan, G. Necula, D. A. Patterson, . Sen, John Wawrzynek, D. Wessel and K. A. Yelick, "The Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View", Technical Report No. UCB/EECS-2008-23, March 21, 2008
- [57] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy "Introduction to the Cell multiprocessor" IBM J. Res. & Dev. vol. 49 no. 4/5 July/September 2005
- [58] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, „Synergistic Processing In Cell’s Multicore Architecture“, Published by the IEEE Computer Society, http://www.research.ibm.com/people/m/mikeg/papers/2006_ieeemicro.pdf
- [59] www.ti.com
- [60] www.intel.com
- [61] www.amd.com
- [62] Sun Niagar processor <http://www.sun.com/processors/niagara/index.jsp>
- [63] Intel TeraScale Cg Research Program: <http://techresearch.intel.com/articles/Tera-Scale/1421.htm>

- [64] www.xilinx.com
- [65] http://www.nvidia.com/object/GPU_Computing.html
- [66] www.streamprocessors.com
- [67] 176x144 Q-Eye chip, www.anafocus.com
- [68] 64x64 C-TON chip www.eutecus.com
- [69] Through Silicon Via
http://www.suss.com/markets/3d_integration/?gclid=CKb1zLj6zZgCFQRptAodSFam1A
- [70] Bump Bonding: <http://www.physics.purdue.edu/vertex/talks/lozano/sld001.htm>
- [71] Video security application: <http://www.objectvideo.com/>

***Appendix:* Description of the cited CNN templates**

Table of content

Average	II
Centroid.....	III
Concave arc filler	IV
Concentric Contour Detector	V
Connected Component Detector (CCD)	VI
Connectivity	VII
Edge Detection	VIII
Halftoning.....	XI
Heat Diffusion	X
Hole finder.....	XI
Hollow	XII
Interpolation	XIII
Logic operators: And, OR	XIV
Mathematical morphology: Erosion, Dilation.....	XV
Patch maker	XVI
Recall.....	XVII
Shadow	XVIII
Skeletonization	XIX
Small killer	XX

Average

$$\mathbf{A} = \begin{array}{|c|c|c|} \hline 0 & 1 & 0 \\ \hline 1 & 2 & 1 \\ \hline 0 & 1 & 0 \\ \hline \end{array} \quad \mathbf{B} = \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \quad z = \begin{array}{|c|} \hline 0 \\ \hline \end{array}$$

I. Global Task

Verbal description: This propagating type template drives a grayscale image to black-and-white. It is similar to a threshold combined with a binary morphological smoothing.

Given: static grayscale image \mathbf{P}

Input: $\mathbf{U(t)}$ = Arbitrary or as a default $\mathbf{U(t)=0}$

Initial State: $\mathbf{X(0) = P}$

Boundary Conditions: Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by $[\mathbf{Y}]=0$

Output: $\mathbf{Y(t) \Rightarrow Y(\infty)}$ = Binary image where black (white) pixels correspond to the locations in \mathbf{P} where the average of pixel intensities over the $r=1$ feedback convolution window is positive (negative).

II. Example: image name: madonna.bmp, image size: 59x59; template name: avertshr.tem .



input



output

Centroid

CENTER1:

$$\mathbf{A}_1 = \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array}$$

$$\mathbf{B}_1 = \begin{array}{|c|c|c|} \hline 1 & 0 & 0 \\ \hline 1 & 4 & -1 \\ \hline 1 & 0 & 0 \\ \hline \end{array}$$

$$z_1 = \boxed{-1}$$

CENTER2:

$$\mathbf{A}_2 = \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array}$$

$$\mathbf{B}_2 = \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 6 & 0 \\ \hline 1 & 0 & -1 \\ \hline \end{array}$$

$$z_2 = \boxed{-1}$$

CENTER3:

$$\mathbf{A}_3 = \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array}$$

$$\mathbf{B}_3 = \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 0 & 4 & 0 \\ \hline 0 & -1 & 0 \\ \hline \end{array}$$

$$z_3 = \boxed{-1}$$

...

CENTER8:

$$\mathbf{A}_8 = \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array}$$

$$\mathbf{B}_8 = \begin{array}{|c|c|c|} \hline 1 & 0 & -1 \\ \hline 1 & 6 & 0 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$$

$$z_8 = \boxed{-1}$$

I. Global Task

Verbal description:

This is a template algorithm. The templates should be sequentially executed one after the other (circularly), as long as the image changes. The algorithm identifies the center point of the black-and-white input object. This is always a point of the object, halfway between the furthestmost points of it.

Given:

static binary image \mathbf{P}

Input:

$\mathbf{U(t)}$ = Arbitrary or as a default $\mathbf{U(t)=0}$

Initial State:

$\mathbf{X(0) = P}$

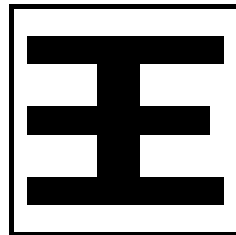
Boundary Conditions:

Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by $[\mathbf{Y}]=0$

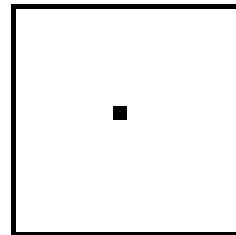
Output:

$\mathbf{Y(t) \Rightarrow Y(\infty)}$ = Binary image where a black pixel indicates the center point of the object in \mathbf{P} .

II. Example: image name: chinese.bmp, image size: 16x16; template name: center.tem .



input



output

Concave arc filler

FILL35:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \\ 1 & 1 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{2}$$

FILL65:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 2 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{3}$$

I. Global Task

Verbal description: This binary-input binary-output propagating type template fills in the concave arcs in the image.

Given: static binary image \mathbf{P}

Input: $\mathbf{U(t)} = \mathbf{P}$

Initial State: $\mathbf{X(0)} = \mathbf{P}$

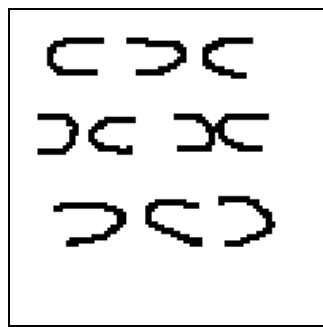
Boundary Conditions: Fixed type, $y_{ij} = -1$ for all virtual cells, denoted by $[\mathbf{Y}] = -1$

Output: $\mathbf{Y(t)} \Rightarrow \mathbf{Y(\infty)}$ = Binary image in which those arcs of objects are filled which have a prescribed orientation.

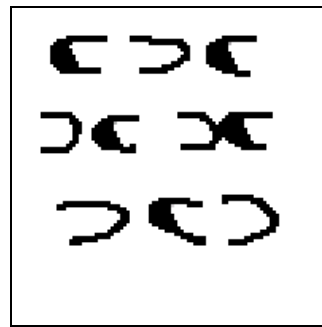
Remark:

In general, the objects of \mathbf{P} that are not filled should have at least 2 pixel wide contour. Otherwise the template may not work correctly.

II. Example: image name: arcs.bmp, image size: 100x100; template name: arc_fill.tem .



input



output ($t=20\tau_{\text{CNN}}$)

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & -2 & 0 & 0 \\ 0 & -4 & 16 & -4 & 0 \\ -2 & 16 & -39 & 16 & -2 \\ 0 & -4 & 16 & -4 & 0 \\ 0 & 0 & -2 & 0 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{0}$$

Concentric Contour Detector

$$\mathbf{A} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 3.5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{-4}$$

I. Global Task

Verbal description: This DTCNN template transforms a black object on a black-and-white image into a number of continuous concentric contours.

Given: static binary image \mathbf{P}

Input: $\mathbf{U(t)} = \mathbf{P}$

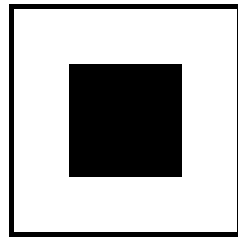
Initial State: $\mathbf{X(0)} = \mathbf{P}$

Boundary Conditions: Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by $[\mathbf{Y}] = 0$

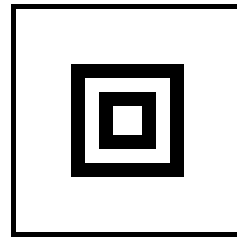
Output: $\mathbf{Y(t)} \Rightarrow \mathbf{Y(\infty)}$ = Binary image representing the concentric black and white rings obtained from \mathbf{P} .

Examples

Example 1: image name: conc1.bmp, image size: 16x16; template name: concont.tem .

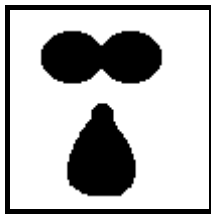


input

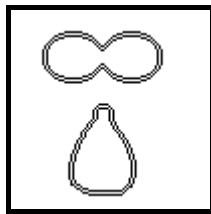


output

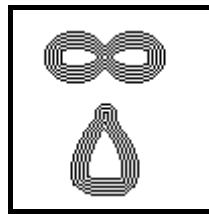
Example 2: image name: conc2.bmp, image size: 100x100; template name: concont.tem .



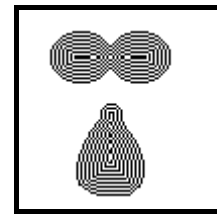
input



output, 3. step



output, 9. step



output, $t = \infty$

Connected Component Detector (CCD)

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 2 & -1 \\ 0 & 0 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{0}$$

I. Global Task

Verbal description: This propagating type binary-input binary-output template shrinks the connected black components to one pixel, and shifts them to one side or one corner of the image.

Given: static binary image \mathbf{P}

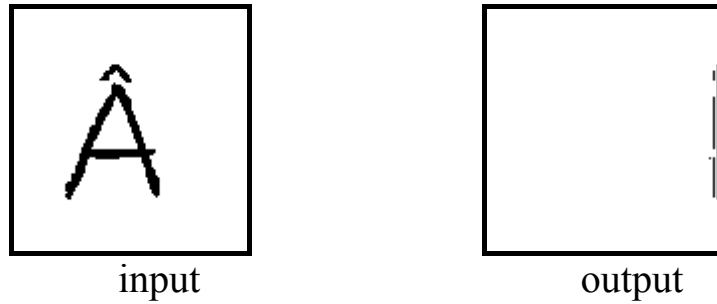
Input: $\mathbf{U(t)}$ = Arbitrary or as a default $\mathbf{U(t)=0}$

Initial State: $\mathbf{X(0)} = \mathbf{P}$

Boundary Conditions: Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by $[\mathbf{Y}]=0$


Output: $\mathbf{Y(t)} \Rightarrow \mathbf{Y(\infty)}$ = Binary image that shows the number of horizontal holes in each horizontal row of image \mathbf{P} .

II. Example: image name: a_letter.bmp, image size: 117x121; template name: ccd_hor.tem .

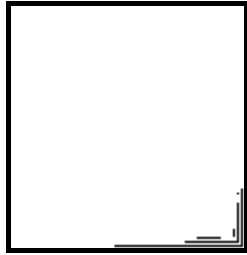


Example2.: Rotated template version

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & -1 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{0}$$



input



output

Connectivity

$$\mathbf{A} = \begin{bmatrix} 0 & 0.5 & 0 \\ 0.5 & 3 & 0.5 \\ 0 & 0.5 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & -0.5 & 0 \\ -0.5 & 3 & -0.5 \\ 0 & -0.5 & 0 \end{bmatrix} \quad z = \boxed{-4.5}$$

I. Global Task

Verbal description: This binary-input binary-output propagating type template compares two almost identical images (input, initial state), and deletes those objects from the initial state, which one's twin object is damaged on the input.

Given: two static binary images \mathbf{P}_1 (mask) and \mathbf{P}_2 (marker). The mask contains some black objects against the white background. The marker contains the same objects, except for some objects being marked. An object is considered to be marked, if some of its black pixels are changed into white.

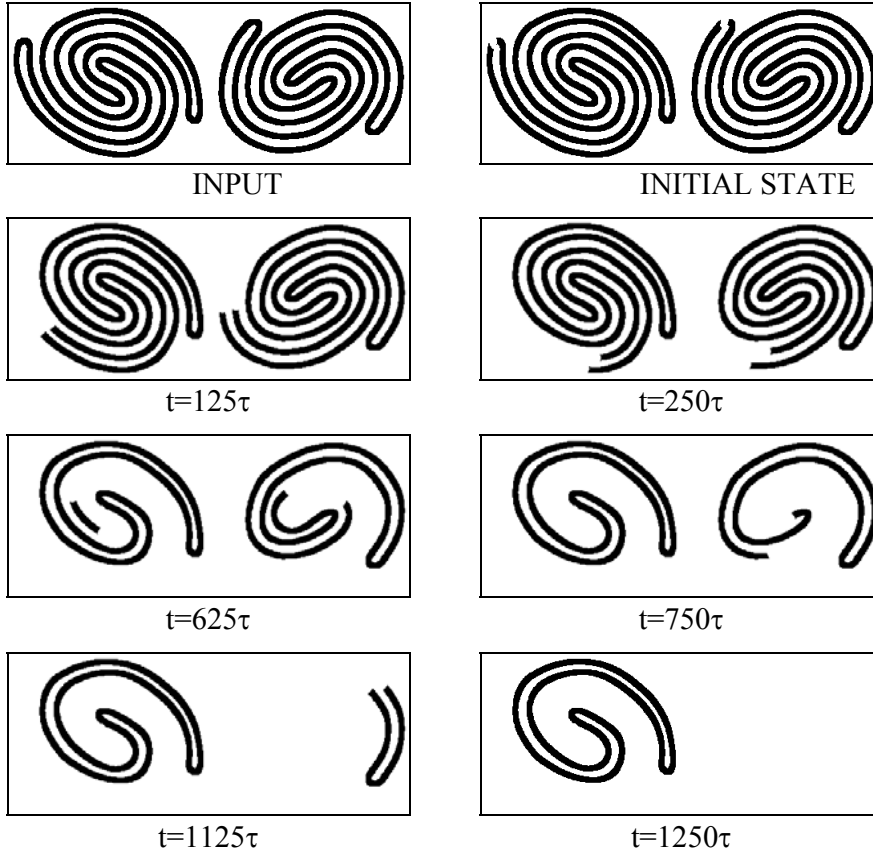
Input: $\mathbf{U}(t) = \mathbf{P}_1$

Initial State: $\mathbf{X}(0) = \mathbf{P}_2$

Boundary Conditions: Fixed type, $u_{ij} = -1$, $y_{ij} = -1$ for all virtual cells, denoted by $[\mathbf{U}] = [\mathbf{Y}] = [-1]$

Output: $\mathbf{Y}(t) \Rightarrow \mathbf{Y}(\infty) = \text{Binary image containing the unmarked objects only.}$

II. Example: image names: connect1.bmp, connect2.bmp; image size: 500x200; template name: connecti.tem .



Edge Detection

$$\mathbf{A} = \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \quad \mathbf{B} = \begin{array}{|c|c|c|} \hline -1 & -1 & -1 \\ \hline -1 & 8 & -1 \\ \hline -1 & -1 & -1 \\ \hline \end{array} \quad z = \boxed{-1}$$

I. Global Task

Verbal description: This binary-input binary-output non-propagating type template extracts the edges of the black object on an image.

Given: static binary image \mathbf{P}

Input: $\mathbf{U(t)} = \mathbf{P}$

Initial State: $\mathbf{X(0)} = \text{Arbitrary}$ (in the examples we choose $x_{ij}(0)=0$)

Boundary Conditions: Fixed type, $u_{ij} = 0$ for all virtual cells, denoted by $[\mathbf{U}]=0$

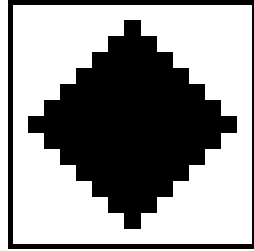
Output: $\mathbf{Y(t)} \Rightarrow \mathbf{Y(\infty)} = \text{Binary image showing all edges of } \mathbf{P} \text{ in black}$

Template robustness: $\rho = 0.12$.

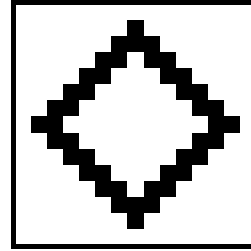
Remark:

Black pixels having at least one white neighbor compose the edge of the object.

II. Examples: image name: logic05.bmp, image size: 44x44; template name: edge.tem.



input



output

Halftoning

$$\mathbf{A} = \begin{bmatrix} -0.07 & -0.1 & -0.07 \\ -0.1 & 1+\epsilon & -0.1 \\ -0.07 & -0.1 & -0.07 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0.07 & 0.1 & 0.07 \\ 0.1 & 0.32 & 0.1 \\ 0.07 & 0.1 & 0.07 \end{bmatrix} \quad z = \begin{bmatrix} 0 \end{bmatrix}$$

I. Global Task

Verbal description: This grayscale input binary-output propagating type template generates a printable black-and-white image, in which the local average is the same as in the original grayscale image.

Given: static grayscale image \mathbf{P}

Input: $\mathbf{U(t)} = \mathbf{P}$

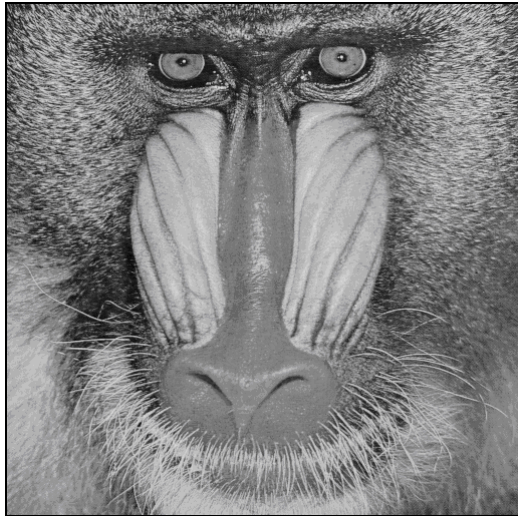
Initial State: $\mathbf{X(0)} = \mathbf{P}$

Boundary Conditions: Fixed type, $u_{ij} = 0, y_{ij} = 0$ for all virtual cells, denoted by $[\mathbf{U}]=[\mathbf{Y}]=0$

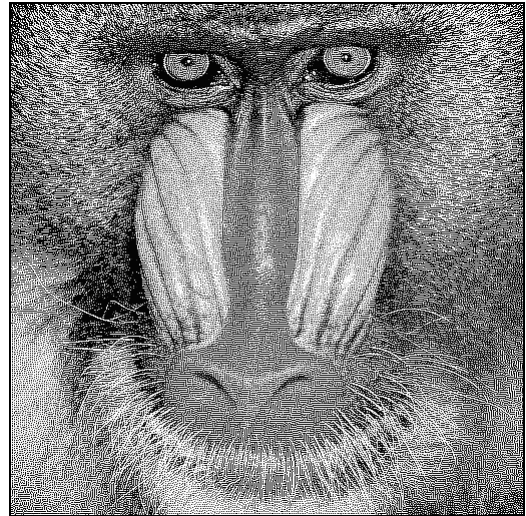
Output: $\mathbf{Y(t)} \Rightarrow \mathbf{Y(\infty)} = \text{Binary image preserving the main features of } \mathbf{P}.$

II. Examples

Example 1: image name: baboon.bmp, image size: 512x512; template name: hlf3.tem .



input



output

Heat Diffusion

$$\mathbf{A} = \begin{bmatrix} 0.1 & 0.15 & 0.1 \\ 0.15 & 0 & 0.15 \\ 0.1 & 0.15 & 0.1 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \begin{bmatrix} 0 \end{bmatrix}$$

I. Global Task

Verbal description: This grayscale-input grayscale-output propagating type template approximates heat diffusion. This has a blurring (out of focus) effect on images.

Given: static noisy grayscale image \mathbf{P}

Input: $\mathbf{U(t)}$ = Arbitrary or as a default $\mathbf{U(t)=0}$

Initial State: $\mathbf{X(0)} = \mathbf{P}$

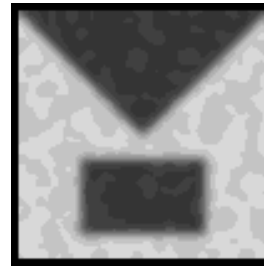
Boundary Conditions: Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by $[\mathbf{Y}]=0$

Output: $\mathbf{Y(t)} \Rightarrow \mathbf{Y(T)}$ = Grayscale image representing the result of the heat diffusion operation.

II. Example: image name: diffus.bmp, image size: 106x106; template name: diffus.tem .



input



output

Hole finder

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 3 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{-1}$$

I. Global Task

Verbal description: This binary-input binary-output propagating type template fills the holes in the image.

Given: static binary image \mathbf{P}

Input: $\mathbf{U(t)} = \mathbf{P}$

Initial State: $\mathbf{X(0)} = \mathbf{1}$ (constant black)

Boundary Conditions: Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by $[\mathbf{Y}] = 0$

Output: $\mathbf{Y(t)} \Rightarrow \mathbf{Y(\infty)}$ = Binary image representing \mathbf{P} with holes filled.

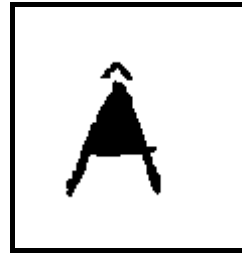
Remark:

- (i) this is a propagating template, the computing time is proportional to the length of the image

II. Example: image name: a_letter.bmp, image size: 117x121; template name: hole.tem .



input



output

Hollow

$$\mathbf{A} = \begin{bmatrix} 0.5 & 0.5 & 0.5 \\ 0.5 & 2 & 0.5 \\ 0.5 & 0.5 & 0.5 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{3.25}$$

I. Global Task

Verbal description: This binary-input binary-output propagating type template fills in the concave areas in or around the objects and finally generates an octagonal bounding box onto them.

Given: static binary image \mathbf{P}

Input: $\mathbf{U(t)} = \mathbf{P}$

Initial State: $\mathbf{X(0)} = \mathbf{P}$

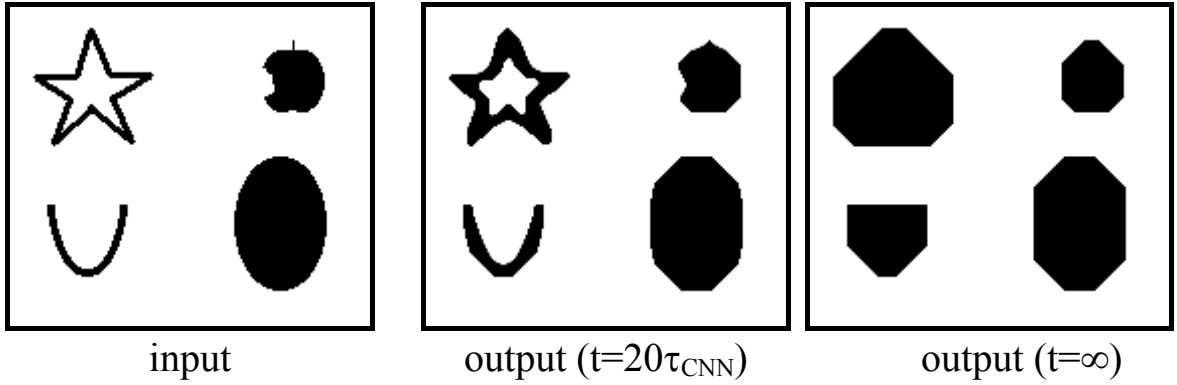
Boundary Conditions: Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by $[\mathbf{Y}] = 0$

Output: $\mathbf{Y(t)} \Rightarrow \mathbf{Y(\infty)}$ = Binary image in which the concave locations of objects are black.

Remark:

In general, the objects of \mathbf{P} that are not filled should have at least a 2-pixel-wide contour. Otherwise the template may not work properly.
The template transforms all the objects to solid black concave polygons with vertical, horizontal and diagonal edges only.

II. Example: image name: hollow.bmp, image size: 180x160; template name: hollow.tem .



Interpolation

I. Global Task

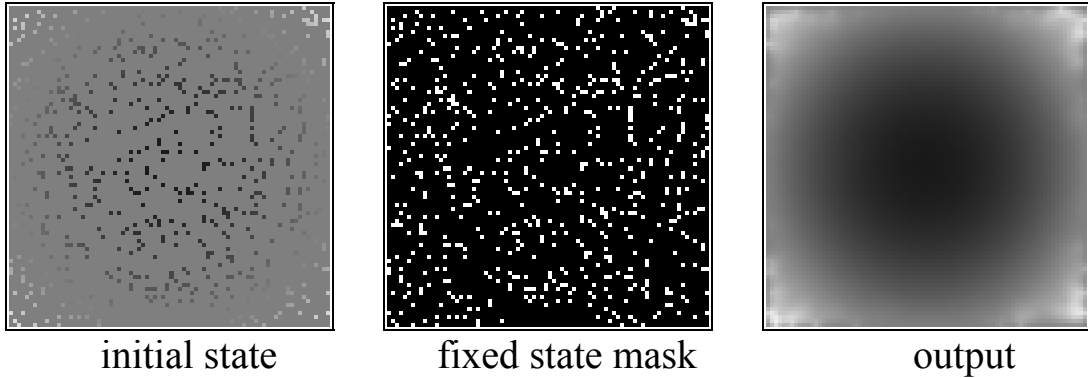
Verbal description:	This grayscale-input grayscale-output propagating type template stretches a smooth surface over a number of given points.
Given:	a static grayscale image \mathbf{P}_1 and a static binary image \mathbf{P}_2
Input:	$\mathbf{U}(\mathbf{t})$ = Arbitrary or as a default $\mathbf{U}(\mathbf{t})=0$
Initial State:	$\mathbf{X}(0) = \mathbf{P}_1$
Fixed State Mask:	$\mathbf{X}_{\text{fix}} = \mathbf{P}_2$
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by $[\mathbf{Y}]=0$
Output:	$\mathbf{Y}(\mathbf{t}) \Rightarrow \mathbf{Y}(\infty)$ = Grayscale image representing an interpolated surface that fits the given points and is as smooth as possible.

Remark:

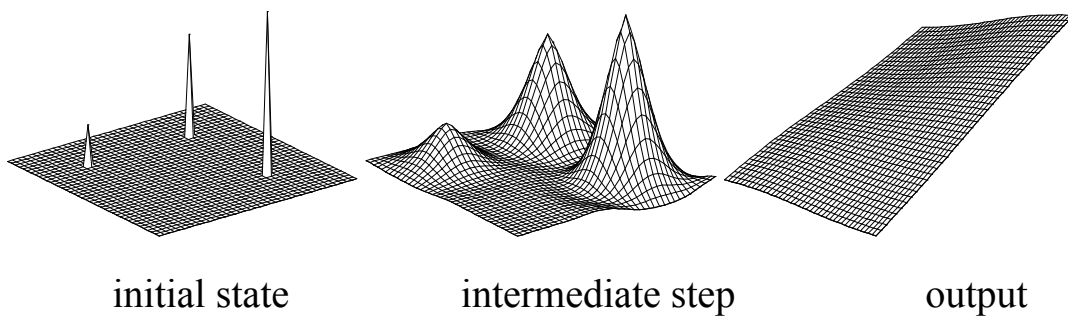
Images \mathbf{P}_1 and \mathbf{P}_2 are to be constructed as follows: if the altitude of the surface to be interpolated is known at point (i,j) , it is preset in \mathbf{P}_1 (state) at the position (i,j) , and the state is kept fixed ($\mathbf{P}_2(i,j) = -1$) during the transient. If the altitude is not known, then zero is filled into the state ($\mathbf{P}_1(i,j) = 0$) and changing of the state is allowed ($\mathbf{P}_2(i,j) = 1$). For exact solution the feedback template must be space variant at the borders. An approximate result can be obtained by using space invariant network. Further information about the space variant network is available in [27] and [29].

II. Examples

Example 1: Ball surface reconstruction (10% of the points is known). Image names: interp1.bmp, interp2.bmp; image size: 80x80; template name: interp.tem .



Example 2: Fitting a surface on three given points. Image size: 80x80.



Logic operators

And, OR

AND:

$\mathbf{A} =$	<table style="border-collapse: collapse; text-align: center;"><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>2</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	2	0	0	0	0	$\mathbf{B} =$	<table style="border-collapse: collapse; text-align: center;"><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	1	0	0	0	0	$z =$	<table style="border-collapse: collapse; text-align: center;"><tr><td>-1</td></tr></table>	-1
0	0	0																						
0	2	0																						
0	0	0																						
0	0	0																						
0	1	0																						
0	0	0																						
-1																								

OR:

$\mathbf{A} =$	<table style="border-collapse: collapse; text-align: center;"><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>2</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	2	0	0	0	0	$\mathbf{B} =$	<table style="border-collapse: collapse; text-align: center;"><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	1	0	0	0	0	$z =$	<table style="border-collapse: collapse; text-align: center;"><tr><td>1</td></tr></table>	1
0	0	0																						
0	2	0																						
0	0	0																						
0	0	0																						
0	1	0																						
0	0	0																						
1																								

I. Global Task

Verbal description: This binary-input binary-output non-propagating type template applies pixel-by-pixel logic AND/OR operation onto two images.

Given: two static binary images \mathbf{P}_1 and \mathbf{P}_2

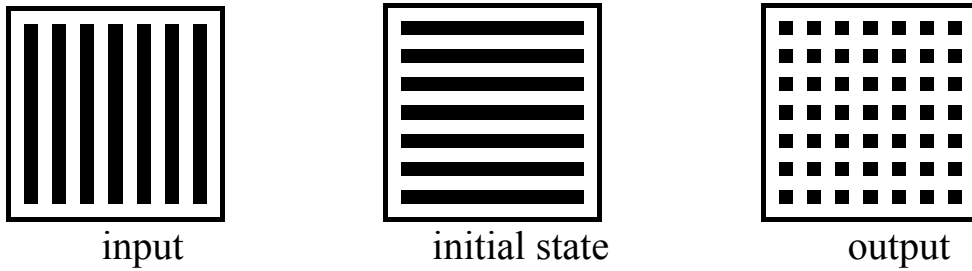
Input: $\mathbf{U}(t) = \mathbf{P}_1$

Initial State: $\mathbf{X}(0) = \mathbf{P}_2$

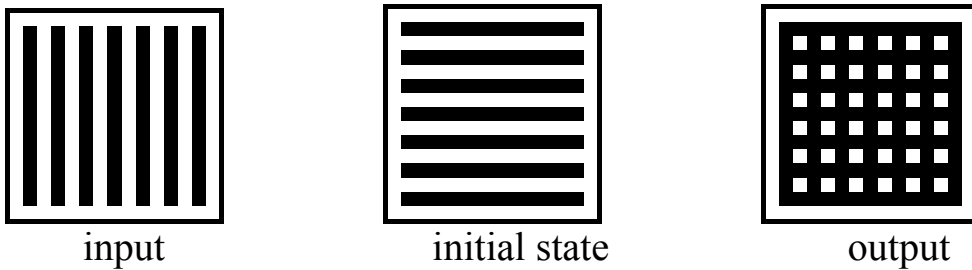
Output: $\mathbf{Y}(t) \Rightarrow \mathbf{Y}(\infty) =$ binary output of the logic operation “AND” between \mathbf{P}_1 and \mathbf{P}_2 . In logic notation, $\mathbf{Y}(\infty) = \mathbf{P}_1 \wedge \mathbf{P}_2$, where \wedge denotes the “conjunction” operator. In set-theoretic notation, $\mathbf{Y}(\infty) = \mathbf{P}_1 \cap \mathbf{P}_2$, where \cap denotes the “intersection” operator.

II. Example: image names: logic01.bmp, logic02.bmp; image size: 44x44; template name: logand.tem / logor.tem.

AND:



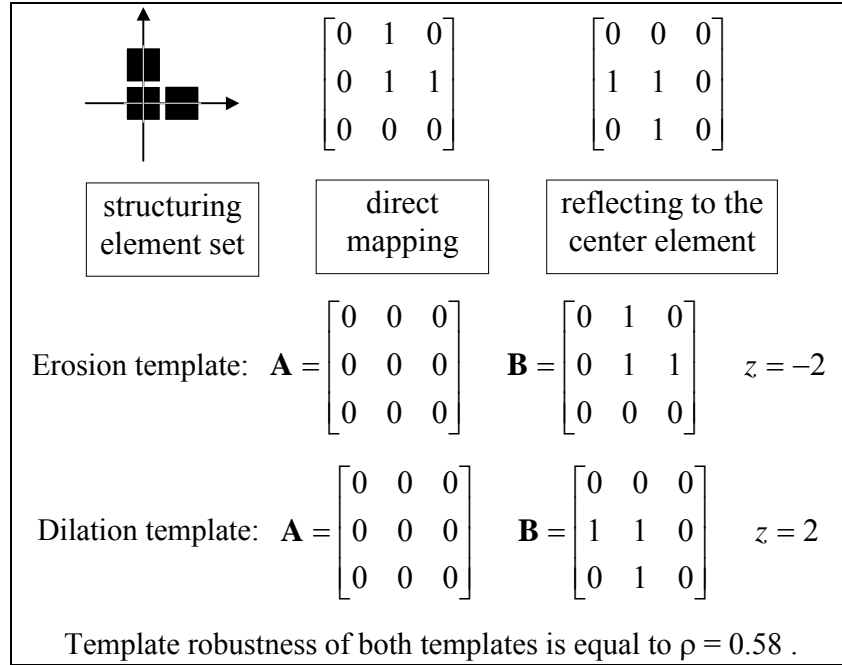
OR:



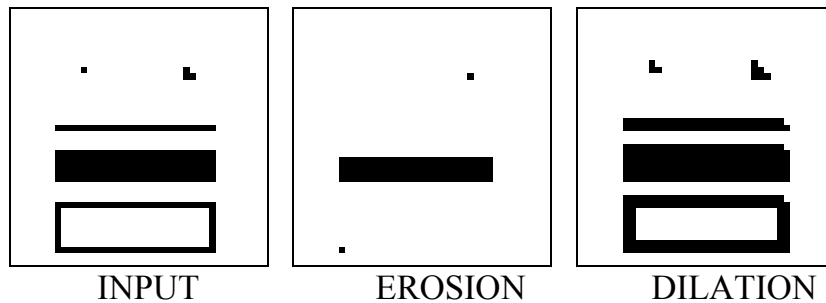
Mathematical morphology:

Erosion, Dilation

The basic operations of binary mathematical morphology are erosion and dilation. These operations are defined by two binary images, one being the operand, the other the structuring element. In the CNN implementation, the former image is the input, while the function (the templates) itself depends on the latter image. If the structuring element set does not exceed the size of the CNN template the dilation and erosion operators can be implemented with a single CNN template. The implementation method is the following: The **A** template matrix is zero in every position. The structuring element set should be directly mapped to the **B** template (See Figure). If it is an erosion operator, the z value is equal to $(1-n)$, where n is the number of 1s in the **B** template matrix. If it is a dilation operator, the **B** template must be reflected to the center element, and the z value is equal to $(n-1)$, where n is the number of 1s in the **B** template matrix. When calculating the operator, the image should be put to the input of the CNN, and the initial condition is zero everywhere. The next Figure shows how to synthesize a template.



Example: Erosion and dilation with the given structuring element set. Image name: binmorph.bmp; image size: 40x40; template names: eros_bin.tem, dilat_bin.tem.



Patch maker

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{4.5}$$

I. Global Task

Verbal description: This binary-input binary-output propagating type template increases the size of the black objects on the image. The expansion is proportional with the running time.

Given: static binary image \mathbf{P}

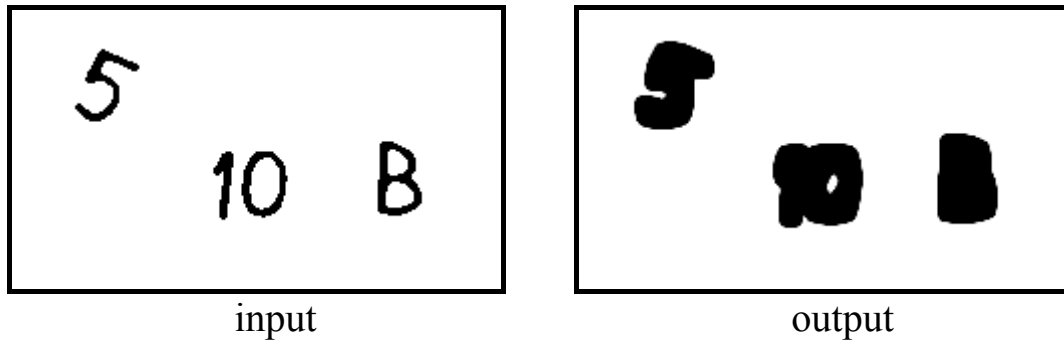
Input: $\mathbf{U(t)} = \mathbf{P}$

Initial State: $\mathbf{X(0)} = \mathbf{P}$

Boundary Conditions: Zero-flux boundary condition (duplicate)

Output: $\mathbf{Y(t)} \Rightarrow \mathbf{Y(T)} =$ Binary image with enlarged objects of the input obtained after a certain time $t = \mathbf{T}$. The size of the objects depends on time \mathbf{T} . When $\mathbf{T} \rightarrow \infty$ all pixels will be driven to black.

II. Example: image name: patchmak.bmp; image size: 245x140; template name: patchmak.tem .



Recall

$$\mathbf{A} = \begin{bmatrix} 0.5 & 0.5 & 0.5 \\ 0.5 & 4 & 0.5 \\ 0.5 & 0.5 & 0.5 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \begin{bmatrix} 3 \end{bmatrix}$$

I. Global Task

Verbal description: This binary-input binary-output propagating type template recalls (reconstructs) the damaged objects. Original image (mask): input. Damaged image (marker) initial state.

Given: two static binary images \mathbf{P}_1 (mask) and \mathbf{P}_2 (marker). \mathbf{P}_2 contains just a part of \mathbf{P}_1 ($\mathbf{P}_2 \subset \mathbf{P}_1$).

Input: $\mathbf{U}(t) = \mathbf{P}_1$

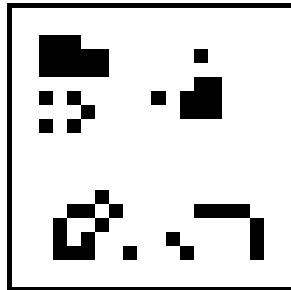
Initial State: $\mathbf{X}(0) = \mathbf{P}_2$

Boundary Conditions: Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by $[Y]=0$

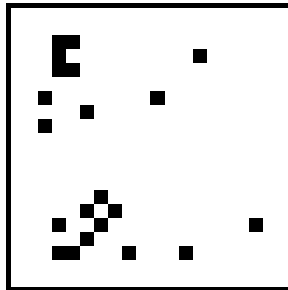
Output: $\mathbf{Y}(t) \Rightarrow \mathbf{Y}(\infty)$ = Binary image representing those objects of \mathbf{P}_1 which are marked by \mathbf{P}_2 .

Template robustness: $\rho = 0.12$.

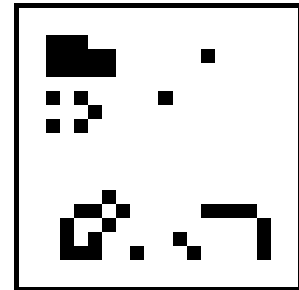
II. Example: image names: figdel.bmp, figrec.bmp; image size: 20x20; template name: figrec.tem



input



initial state



output

Shadow

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 2 \\ 0 & 0 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \begin{bmatrix} 0 \end{bmatrix}$$

I. Global Task

Verbal description: This binary-input binary-output propagating type template generates the shadow of the objects.

Given: static binary image \mathbf{P}

Input: $\mathbf{U(t)} = \mathbf{P}$

Initial State: $\mathbf{X(0)} = \mathbf{1}$

Boundary Conditions: Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by $[\mathbf{Y}] = 0$

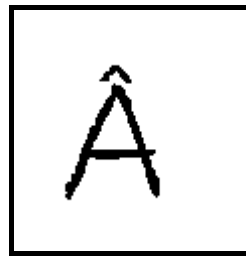
Output: $\mathbf{Y(t)} \Rightarrow \mathbf{Y(\infty)} =$ Binary image representing the left shadow of the objects in \mathbf{P} .

Remark:

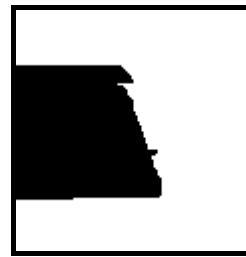
By modifying the position of the off-center \mathbf{A} template element the template can be sensitized to other directions as well.

II. Examples

Example 1: Left shadow. Image name: a_letter.bmp, image size: 117x121; template name: shadow.tem .



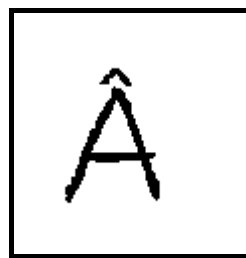
input



output

Example2.: Rotated template version

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \begin{bmatrix} 0 \end{bmatrix}$$



input



output

Skeletonization

Task description and algorithm

The algorithm finds the skeleton of a black-and-white object. The 8 templates should be applied circularly, always feeding the output back to the input before using the next template [19].

The templates of the algorithm:

SKELBW1:

$$\mathbf{A}_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \mathbf{B}_1 = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad z_1 = \boxed{-1}$$

SKELBW2:

$$\mathbf{A}_2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \mathbf{B}_2 = \begin{bmatrix} 2 & 2 & 2 \\ 0 & 9 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad z_2 = \boxed{-2}$$

SKELBW3:

$$\mathbf{A}_3 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \mathbf{B}_3 = \begin{bmatrix} 0 & 1 & 1 \\ -1 & 5 & 1 \\ 0 & -1 & 0 \end{bmatrix} \quad z_3 = \boxed{-1}$$

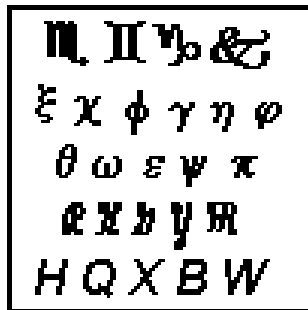
...

SKELBW8:

$$\mathbf{A}_8 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \mathbf{B}_8 = \begin{bmatrix} 2 & 0 & -1 \\ 2 & 9 & -2 \\ 2 & 0 & -1 \end{bmatrix} \quad z_8 = \boxed{-2}$$

The robustness of templates SKELBW1 and SKELBW2 are $\rho(\text{SKELBW1}) = 0.18$ and $\rho(\text{SKELBW2}) = 0.1$, respectively. Other templates are the rotated versions of SKELBW1 and SKELBW2, thus their robustness values are equal to the mentioned ones.

Example: image name: skelbwi.bmp, image size: 100x100; template names: skelbw1.tem, skelbw2.tem, ..., skelbw8.tem.



input



output

Small killer

$$\mathbf{A} = \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 2 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array} \quad \mathbf{B} = \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \quad z = \begin{array}{|c|} \hline 0 \\ \hline \end{array}$$

I. Global Task

Verbal description: This binary-input binary-output propagating type template deletes the small black objects, and smoothens the boundary of the larger ones.

Given: static binary image \mathbf{P}

Input: $\mathbf{U(t)} = \mathbf{P}$

Initial State: $\mathbf{X(0)} = \mathbf{P}$

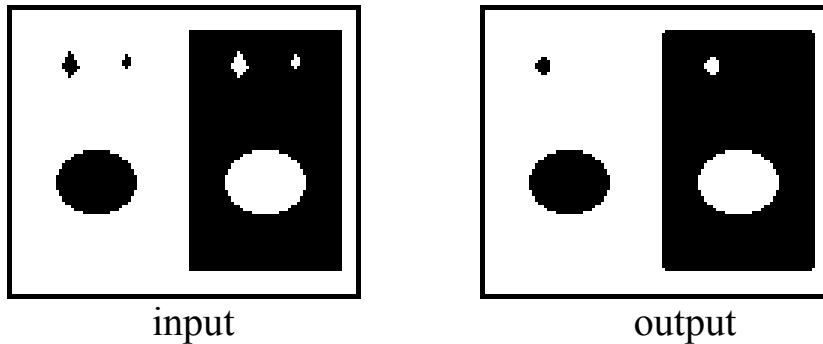
Boundary Conditions: Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by $[\mathbf{Y}] = 0$

Output: $\mathbf{Y(t)} \Rightarrow \mathbf{Y(\infty)} =$ Binary image representing \mathbf{P} without small objects.

Remark:

This template drives dynamically white all those black pixels that have more than four white neighbors, and drives black all white pixels having more than four black neighbors.

II. Example: image name: smkiller.bmp; image size: 115x95; template name: smkiller.tem .



Example: image names: LenaS.bmp; image size: 128x128; template name: CS2.tem.